软·件·设·计·技·术

Java 设计模式深入研究

刘德山 金百东 ◎ 编著

人民邮电出版社
POSTS & TELECOM PRESS

Java 设计模式深入研究

封面设计：董志桢

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-35186-9

ISBN 978-7-115-35186-9

定价：45.00 元

软·件·设·计·技·术

Java 设计模式深入研究

刘德山 金百东 ◎ 编著

人民邮电出版社
北京

图书在版编目（CIP）数据

Java设计模式深入研究 / 刘德山，金百东编著. -- 北京 : 人民邮电出版社，2014.7
ISBN 978-7-115-35186-9

Ⅰ. ①J… Ⅱ. ①刘… ②金… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第087800号

内 容 提 要

设计模式是一套被重复使用的代码设计经验的总结。本书面向有一定 Java 语言基础和一定编程经验的读者，旨在培养读者良好的设计模式思维方式，加强对面向对象思想的理解。

全书共分 12 章，首先强调了接口和抽象类在设计模式中的重要性，介绍了反射技术在设计模式中的应用。然后，从常用的 23 个设计模式中精选 10 个进行了详细的讲解，包括 2 个创建型模式、4 个行为型模式、4 个结构型模式。本书理论讲解透彻，应用示例深入。设计模式的讲解均从生活中的一类常见事物的分析引出待讨论的主题，然后深入分析设计模式，最后进行应用探究。应用探究部分所有示例都源自应用项目，内容涉及 Java、JSP、JavaScript、Ajax 等实用技术，知识覆盖面广。

本书可供高等院校计算机相关专业本科生和研究生设计模式、软件体系结构等课程使用，对高级程序员、软件工程师、系统架构师等专业研究人员也具有一定的参考价值。

◆ 编　　著　刘德山　金百东
责任编辑　邹文波
责任印制　彭志环　焦志炜
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京中新伟业印刷有限公司印刷
◆ 开本：787×1092　1/16
印张：13.75　　2014 年 7 月第 1 版
字数：363 千字　　2014 年 7 月北京第 1 次印刷

定价：45.00 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315

前　言

关于设计模式

模式是从不断重复出现的事件中发现和抽象出的规律，是解决问题形成的经验总结。设计模式作为一种模式，最早应用于建筑领域，目的是在图纸上以一种结构化、可重用化的方法，获得建筑的基本要素。渐渐地，这种思想在软件领域流行起来，并获得发展，形成了软件开发的设计模式。

软件设计模式被认为是一套被反复使用、多数人知晓、经过分类编目的代码设计经验的总结。最早的设计模式是由 GOF 在《Design Patterns: Elements of Reusable Object-Oriented Software》一书提出的，这也被称为经典设计模式，共有 23 个，分为创建型模式、行为型模式、结构型模式三类。使用设计模式的目的是为了提高代码的可重用性、让代码更容易被他人理解、系统更加可靠。

创作背景

应用设计模式构建有弹性、可扩展的应用系统已成为软件人员的共识，越来越多的程序员需要掌握设计模式的内容。近年来，市场上也涌现了一些有关设计模式的书籍。这些书籍各有特点，多从生活中的示例入手，让读者对所述设计模式有一定的感性认识，然后引入设计模式概念，最后用计算机专业程序进行理性说明。通常，示例部分内容成熟，但专业应用部分的讲解稍显单薄，笔者认为主要有以下几点。第一，示例偏简单，读者看过一遍就理解其含义，但很难会真正应用；第二，示例趣味性不足，例如，很多讲解基于命令行界面，而实际应用更多的是图形界面；第三，有些书以 ERP 各个具体模块讲解设计模式，显得过于单一。上述原因是促使我们写作本书的动力。

本书内容

本书首先利用两章讲解了用到的预备知识：接口与抽象类，反射。然后从常用的 23 个设计模式中精选了 10 个进行讲解，包括 2 个创建型模式：工厂、生成器模式，4 个行为型模式：观察者、访问者、状态、命令模式，4 个结构型模式：桥接、代理、装饰器、组合模式。每个模式一般都包含以下四部分。

（1）问题的提出：一般从生活中的一类常见事物引出待讨论的主题。

（2）模式讲解：用模式方法解决与之对应的最基本问题，归纳出角色及 UML 类图。

（3）深入理解模式：讲解笔者对模式的一些体会。

（4）应用探究：均是实际应用中较难的程序，进行了详细的问题分解、分析与说明。

本书特色

（1）示例丰富，讲解细致，有命令行程序，也有图形界面、Web 程序等，涉及 Java、JSP、JavaScript、Ajax 等技术。

（2）强调了语义的作用。一方面把设计模式抽象转化成日常生活中最朴实的语言；另一方面把生活中对某事物“管理”的语言转译成某设计模式。相比而言，后者更为重要。

（3）强调了反射技术的作用。对与反射技术相关的设计模式均做了详细的论述。

（4）提出了如何用接口思维巧妙实现 C++标准模板库方法功能的技术手段。

学习设计模式方法

（1）在清晰设计模式基础知识的基础上，认真实践应用探究中的每一个示例，并充分分析，加以思考。

（2）学习设计模式不是一朝一夕的事，不能好高骛远。它是随着读者思维的发展而发展的，一定要在项目中亲身实践，量变引起质变，有句话说得好：“纸上得来终觉浅，决知此事要躬行”。

（3）加强基础知识训练，如数据结构、常用算法等。基础知识牢固了，学习任何新事物都不会发慌，有信心战胜它。否则，知识学得再多，也只是空中楼阁。

（4）不要为了模式而模式，要在项目中综合考虑，统筹安排。

关于本书的使用

本书提供各章节的完整代码供读者使用。

（1）由于篇幅关系，大多数程序的导入（import）命令在书中的示例中没有给出，这些语句需要读者自行加入。但给出的程序源码是完整的（包含导入命令）。

（2）在使用反射时，例如，使用 XML 文件或 properties 文件封装类的配置信息时，如果被封装的类在一个包中，应在配置文件中类的名字之前指明该类所属的包，这样程序才能顺利编译。

（3）Web 程序使用 Tomcat 服务器，版本是 Tomcat 7.0。

（4）部分章节需要连接数据库，本书的数据库采用 MySQL 5.5。数据库操作需要 MySQL 驱动程序，本书使用的是 connection-java-5.1.17-bin.jar，可从 MySQL 官网（http://dev.mysql.com/downloads/ connector/）下载。之后需要添加到项目的 classpath 环境变量中；如果是 Java Web 应用，应将驱动程序复制到 Web 项目的 WEB-INF/lib 文件夹中。

（5）连接数据库的工具类 DbProc 位于第 4 章，全书通用。如果需要，复制到相应的项目中即可。

（6）本书使用的数据库名字是 test，包括 login、student、teacher、score 等表。建立表的 SQL 命令在源码的文件 create.txt 中。

总之，设计模式是一门重要的计算机软件开发技术，笔者希望尽一些微薄之力，为我国的设计模式研究添砖加瓦。但由于水平有限，时间紧迫，书中难免有不妥或疏漏之处，恳请广大读者批评指正。

编　者

2014 年 3 月

目　录

第 1 章　接口与抽象类 …… 1

1.1　语义简单描述 …… 1
1.2　与框架的关系 …… 2
1.3　拓展研究 …… 6
1.3.1　柔性多态 …… 6
1.3.2　借鉴 STL 标准模板库 …… 9

第 2 章　反射 …… 12

2.1　反射的概念 …… 12
2.2　统一形式调用 …… 12
2.3　反射与配置文件 …… 16
2.3.1　反射与框架 …… 16
2.3.2　Properties 配置文件 …… 17

第 3 章　工厂模式 …… 20

3.1　问题的提出 …… 20
3.2　简单工厂 …… 21
3.2.1　代码示例 …… 21
3.2.2　代码分析 …… 22
3.2.3　语义分析 …… 23
3.3　工厂 …… 24
3.3.1　代码示例 …… 24
3.3.2　代码分析 …… 25
3.4　抽象工厂 …… 26
3.4.1　代码示例 …… 26
3.4.2　代码分析 …… 27
3.4.3　典型模型语义分析 …… 28
3.4.4　其他情况 …… 28
3.5　应用探究 …… 30
3.6　自动选择工厂 …… 36

第 4 章　生成器模式 …… 38

4.1　问题的提出 …… 38
4.2　生成器模式 …… 40
4.3　深入理解生成器模式 …… 42
4.4　应用探究 …… 45

第 5 章　观察者模式 …… 59

5.1　问题的提出 …… 59
5.2　观察者模式 …… 59
5.3　深入理解观察者模式 …… 61
5.4　JDK 中的观察者设计模式 …… 67
5.5　应用探究 …… 71

第 6 章　桥接模式 …… 81

6.1　问题的提出 …… 81
6.2　桥接模式 …… 82
6.3　深入理解桥接模式 …… 84
6.4　应用探究 …… 88

第 7 章　代理模式 …… 98

7.1　问题的提出 …… 98
7.2　代理模式 …… 98
7.3　虚拟代理 …… 99
7.4　远程代理 …… 104
7.4.1　RMI 通信 …… 105
7.4.2　RMI 代理模拟 …… 107
7.5　计数代理 …… 109
7.6　动态代理 …… 112
7.6.1　动态代理的成因 …… 112
7.6.2　自定义动态代理 …… 112
7.6.3　JDK 动态代理 …… 115

第 8 章　状态模式 …… 118

8.1　问题的提出 …… 118
8.2　状态模式 …… 118
8.3　深入理解状态模式 …… 120

8.4 应用探究 …… 125

第 9 章 访问者模式 …… 137

9.1 问题的提出 …… 137
9.2 访问者模式 …… 137
9.3 深入理解访问者模式 …… 140
9.4 应用探究 …… 145

第 10 章 命令模式 …… 156

10.1 问题的提出 …… 156
10.2 命令模式 …… 156
10.3 深入理解命令模式 …… 158
10.3.1 命令集管理 …… 158
10.3.2 加深命令接口定义的理解 …… 160
10.3.3 命令模式与 JDK 事件处理 …… 162
10.3.4 命令模式与多线程 …… 165
10.4 应用探究 …… 168

第 11 章 装饰器模式 …… 176

11.1 问题的提出 …… 176
11.2 装饰器模式 …… 177
11.3 深入理解装饰器模式 …… 179
11.3.1 具体构件角色的重要性 …… 179
11.3.2 JDK 中的装饰模式 …… 180
11.4 应用探究 …… 182

第 12 章 组合模式 …… 195

12.1 问题的提出 …… 195
12.2 组合模式 …… 197
12.3 深入理解组合模式 …… 199
12.3.1 其他常用操作 …… 199
12.3.2 节点排序 …… 201
12.4 应用探究 …… 202

参考文献 …… 214

第1章　接口与抽象类

1.1　语义简单描述

接口与抽象类是面向对象思想的两个重要概念。接口仅是方法定义和常量值定义的集合，方法没有函数体；抽象类定义的内容理论上比接口中的内容要多得多，可定义普通类所包含的所有内容，还可定义抽象方法，这也正是叫作抽象类的原因所在。接口、抽象类本身不能示例化，必须在相应子类中实现抽象方法，才能获得应用。

那么，如何更好地理解接口与抽象类？接口中能定义抽象方法，为什么还要抽象类？抽象方法无函数体，不能示例化，说明接口、抽象类本身没有用途，这种定义有意义吗？接口与抽象类的关系到底如何？

获得这些问题答案的最好办法就是来自于生活实践。例如写作文的时候，一定要先思考好先写什么，后写什么；做几何题的时候，要想清楚如何引辅助线，用到哪些公理、定理等；做科研工作的时候，一定要思索哪些关键问题必须解决；工厂生产产品前，必须制订完善的生产计划等。也就是说，人们在做任何事情前，一般来说是先想好，再去实现。这种模式在生活中是司空见惯的。因此，Java 语言一定要反映“思考 - 实现”这一过程，通过不同关键字来实现，即用接口（interface）、抽象类（abstract）来反映思考阶段，用子类（class）来反映实现阶段。

从上文易于得出：“思考 - 实现”是接口、抽象类的简单语义。从此观点出发，结合生活实际，可以方便回答许多待解答的问题，具体如下。

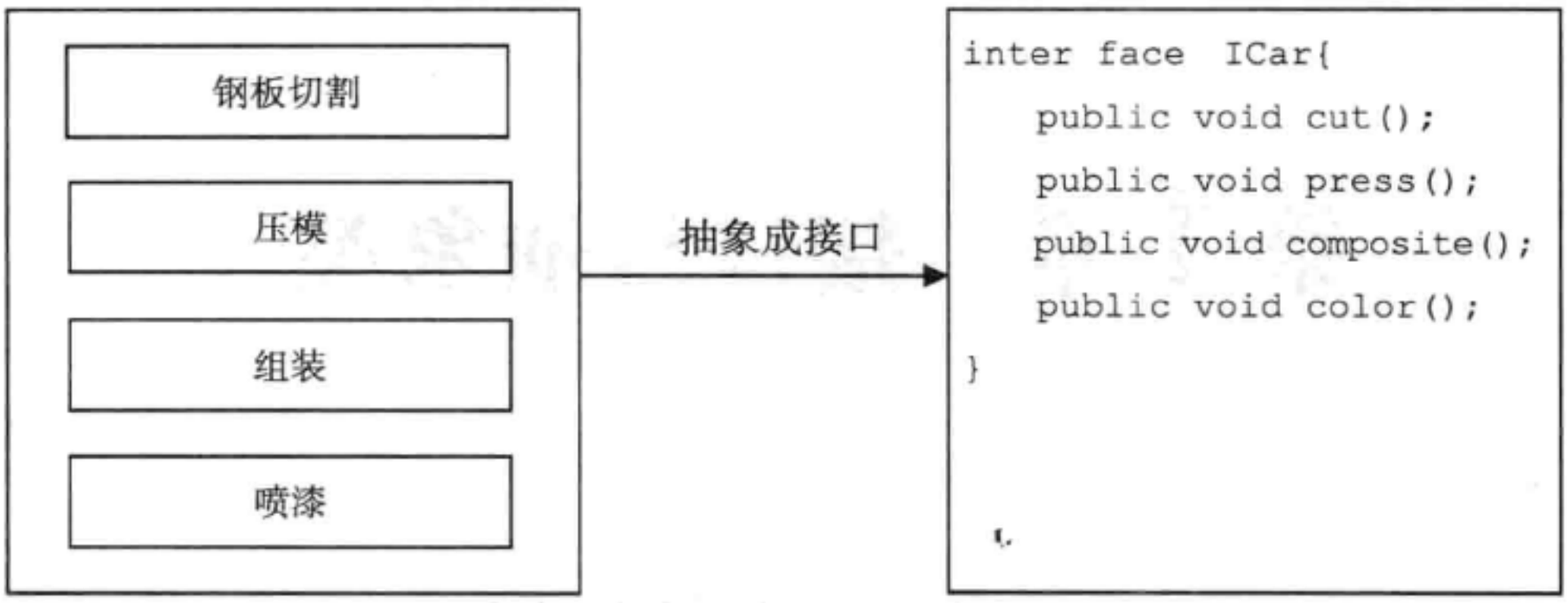
为什么接口、抽象类不能示例化？由于接口、抽象类是思考的结果，只是提出了哪些问题需要解决，无需函数体具体内容，当然不能示例化了。

为什么接口、抽象类必须由子类实现？提出问题之后，总得有解决的方法。Java 语言是通过子类来实现的，当然一定要解决“思考”过程中提出的所有问题。

接口、抽象类有什么区别？可以这样考虑，人类经思考后提出的问题一般有两类：一类是“顺序”问题，另一类是“顺序+共享”问题。前者是用接口描述的，后者是用抽象类来描述的。

图 1-1 所示为一个生产小汽车具体的接口示例。

图 1-1（a）所示为生产小汽车可由钢板切割、压模、组装、喷漆 4 个工序组成。这些工序是顺序关系，因此转化成接口是最恰当的，如图 1-1（b）所示。



（a）小汽车生产抽象工序　　（b）转化成的接口

图 1-1　生产小汽车接口示例

抽象类与接口不同，假设要组装多种价位的电脑，其配置参数如图 1-2 所示。

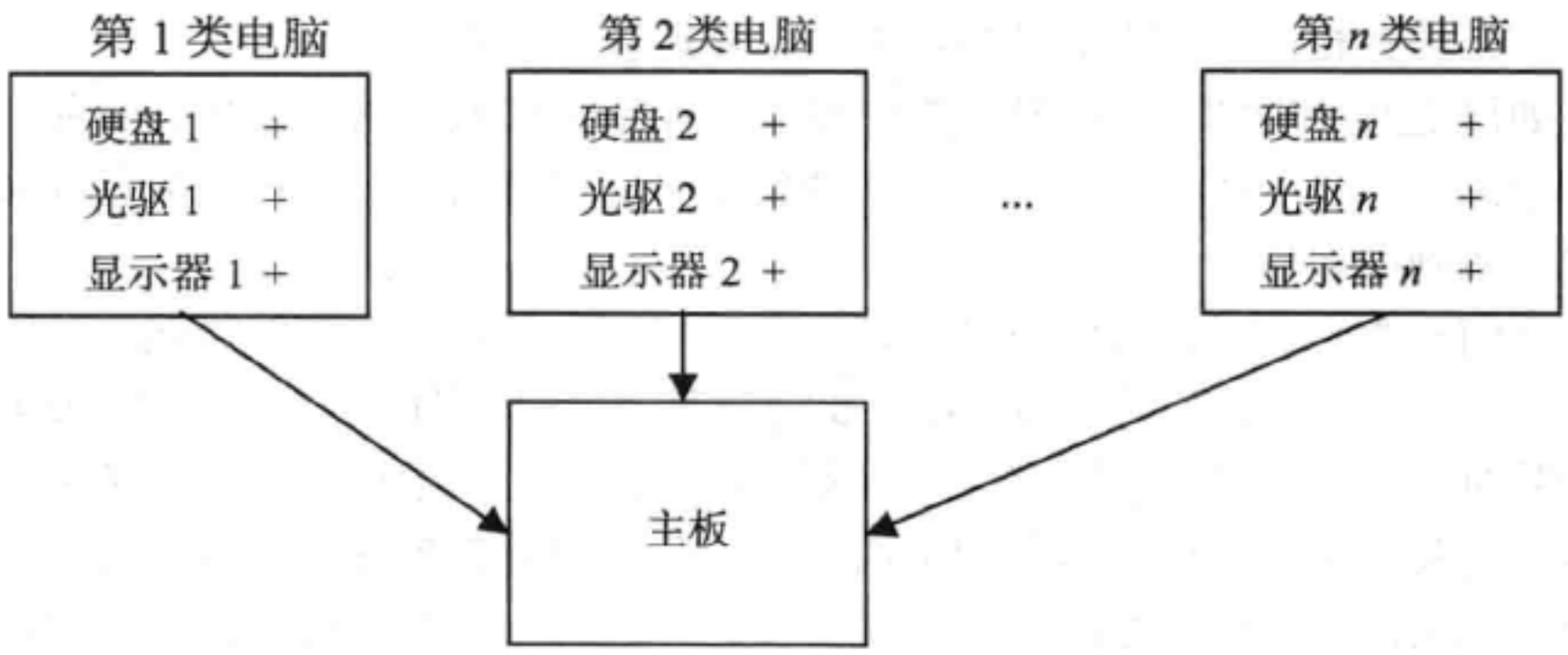


图 1-2　电脑抽象类示例

可以看出，要配置 n 种电脑。每种电脑的硬盘、光驱、显示器是不同类型的，属于并列结构（也可以看作顺序结构）；主板是相同类型的，属于共享结构。因此，转化成的抽象类如下。

```
abstract class Computer{
    abstract void makeHarddisk();        //表明每类电脑有不同的硬盘
    abstract void makeOptical();         //表明每类电脑有不同的光驱
    abstract void makeMonitor();         //表明每类电脑有不同的显示器
    void makeMainBoard(){ };             //表明所有类型的电脑有相同类型的主板
}
```

1.2　与框架的关系

通过 1.1 节的描述可以看出，接口、抽象类代表了提出的两类问题的抽象字符描述。这是理解接口、抽象类的基础，是编制框架的关键。框架包括方法框架与流程框架。下面通过

示例加以说明。

【例 1-1】 方法框架示例。编制求对象数组最大值的泛型方法。

```
//ILess.java:定义二元比较方法
public interface ILess<T> {
    boolean less(T x, T y);
}

//Algo.java:泛型方法类
public class Algo<T> {
    public T getMax(T t[], ILess<T> cmp){
        T maxValue = t[0];
        for(int i=1; i<t.length; i++){
            if(cmp.less(maxValue, t[i]))        //这一行是理解的关键
                maxValue = t[i];
        }
        return maxValue;
    }
}
```

（1）求对象数组的最大值算法比较简单，这里就不多言了。ILess 接口定义了二元比较方法，getMax()是求对象数组最大值的泛型方法。可以发现，根本无须实现 ILess 的子类，上述框架程序即编译成功。

（2）框架程序若获得具体应用，则必须实现 ILess 的子类。以求整型数组最大值及学生成绩最大值加以说明，代码如下。

```
//InteLess.java: 整型数比较器
public class InteLess implements ILess<Integer> {
    public boolean less(Integer x, Integer y) {
        return x<y;
    }
}

//Student.java: 学生基本类
public class Student {
    String name; //姓名
    int grade;   //成绩
    public Student(String name, int grade){
        this.name = name;
        this.grade= grade;
    }
}

//StudLess.java: 学生成绩比较器
public class StudLess implements ILess<Student> {
    public boolean less(Student x, Student y) {
        return x.grade < y.grade;
    }
}

//Test.java: 测试类
```

```
public class Test {
    public static void main(String[] args ) {
        Algo<Integer> obj = new Algo();
        ILess<Integer> cmp = new InteLess();
        Integer a[] = {3,9,2,8};
        Integer max = obj.getMax(a, cmp);
        System.out.println("Integer max=" +max);

        Algo<Student> obj2 = new Algo();
        ILess<Student> cmp2 = new StudLess();
        Student s[]={new Student("li",70),new Student("sun",90),
                     new Student("zhao",80)};
        Student max2 = obj2.getMax(s, cmp2);
        System.out.println("Student max grade:" + max2.grade);
    }
}
```

（3）根据上文可以看出，要实现求某类对象数组的最大值，主要工作是编制具体的从 ILess 接口派生的子类代码，重写 less()方法，自定义比较规则即可。

【例 1-2】 流程框架示例：求圆、矩形的面积。要求：当求圆面积时，能输入圆的半径；当求矩形面积时，能输入长、宽。

分析：根据题目特征得出，对不同的形状有不同的输入参数，有不同的求面积算法。“输入”及“求面积”功能是并列关系，因此，用接口来定义“输入”及“求面积”功能是最恰当的。由此接口出发，得到的相关类代码如下。

```
//IShape.java : 形状接口定义
public interface IShape {
    boolean input();                    //输入方法
    float getArea();                    //求面积方法
}

//ShapeProc.java: 流程处理类
public class ShapeProc {
    private IShape shape;
    public ShapeProc(IShape shape){
        this.shape = shape;
    }

    public float process(){             //每个形状处理包括输入及求面积两步
        shape.input();                  //输入功能
        float value = shape.getArea(); //求面积功能
        return value;                   //返回面积
    }
}
```

（1）ShapeProc 是对接口多态对象的封装类。process()方法表明了对某形状的统一处理过程，包括参数输入 input()方法及求面积 getArea()方法。可以发现，根本无需实现 IShape 的子类，上述流程框架程序即编译成功。

（2）流程框架程序若获得具体应用，则必须实现 IShape 的子类，圆类、矩形类及测试类代码如下。

```
//Circle.java:圆类
import java.util.*;
public class Circle implements IShape {
    float r;
    public float getArea() {
        float s = (float)Math.PI*r*r;
        return s;
    }
    public boolean input() {
        System.out.println("请输入半径:");
        Scanner s = new Scanner(System.in);
        r = s.nextFloat();
        return true;
    }
}
//Rect.java:矩形类
import java.util.*;
public class Rect implements IShape {
    float width,height;
    public float getArea() {
        float s = width*height;
        return s;
    }
    public boolean input() {
        System.out.println("请输入宽、高:");
        Scanner s = new Scanner(System.in);
        width = s.nextFloat();
        height = s.nextFloat();
        return true;
    }
}
//Test.java:测试类
public class Test{
    public static void main(String[] args) {
        IShape shape = new Circle();
        ShapeProc obj = new ShapeProc(shape);
        float value = obj.process();
        System.out.println("圆面积:" + value);

        IShape shape2 = new Rect();
        ShapeProc obj2 = new ShapeProc(shape);
        float value2 = obj.process();
        System.out.println("矩形面积:" + value);
    }
}
```

（3）根据上文可以看出，要实现求某形状的面积，主要工作是编制具体的从 IShape 接口派生的子类代码，重写 input()、getArea()方法，而流程代码无需重写，共享在 ShapeProc 类中的 process()方法中了。

通过例 1-1、例 1-2 可以得出一个重要的框架编程原则：面向接口、抽象类进行编程。也就是说，只要接口、抽象类是稳定的，一般可以抛开具体的实现子类进行编程，容易形成一个稳定的框架系统。

1.3 拓 展 研 究

1.3.1 柔性多态

1. 问题提出

仍以求圆和长方形面积为例。假设其类图如图 1-3 所示。

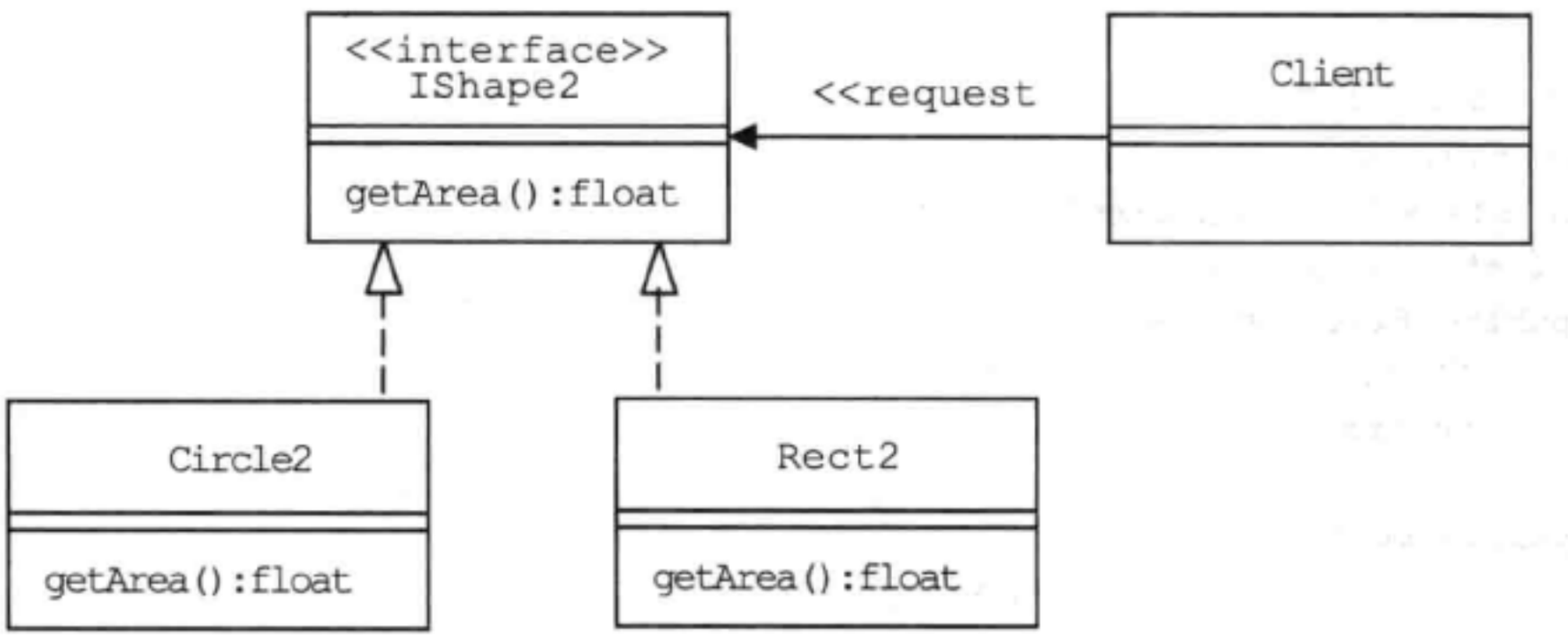


图 1-3 圆和长方形面积功能类图

可以看出这是常规的多态程序设计，父类是 IShape，多态接口函数是 float getArea()；子类 Circle2 及 Rectangle 分别重写了多态函数 getArea()；客户端通过动态绑定对接口编程实现了求圆或长方形面积的功能。

但是如果随着时间的推移还需要求圆和长方形的周长，该如何修改程序呢?

普通的思路是：重新定义接口 IShape2，增加求周长接口函数 float getPerimeter()， 再在 Circle2 及 Rect2 类中实现 getPerimeter()函数的具体功能。这势必造成接口及实现模块、客户端程序都需要修改并重新编译。这是我们不希望看到的，我们希望仅底层具体模块功能可以修改并编译，而接口、上层模块及客户端程序不要重新编写。

因此，如何巧妙运用多态满足不断变化需求分析的需要，只修改需要改变的具体模块，其他模块不修改，是即将论述的中心内容。

普通多态编程局限性：如果接口函数内容发生变化，那么相应的各实现子类必须发生变化，导致相关联的各级模块必须重新编程及编译，这即是普通多态编程的局限性。造成这一结果的主要原因是父类、子类定义的多态函数关联过强，消除这种关联性是实现柔性多态功能的关键。

2. 柔性多态代码示例

柔性多态是指程序架构必须满足不断发展的需求分析的需要，只需修改需要改变的子模块，而相关联模块及主程序都不需要变化。以求圆和长方形面积、周长为例，采用柔性多态，

具体代码如下。

```
//IShape2.java:定义柔性多态接口
public interface  IShape2 {
    //多态函数定义
    public Object dispatch(int nID,Object in);
}

//Circle2.java:圆类
class Circle2  implements IShape2
{
    public float r;
    public Circle2(float r){
        this.r = r;
    }
//多态方法
    public Object dispatch(int nID,Object in){
        Object obj=null;
        switch(nID){
        case 0:
            obj = getArea(in);break;
        case 1:
            obj = getPerimeter(in);break;
        }
        return obj;
    }
Object getArea(Object in){                          //非多态方法
        float area = (float)Math.PI*r*r;
        return new Float(area);
    }
    Object getPerimeter(Object in){                 //非多态方法
        float len = (float)Math.PI*r*2.0f;
        return new Float(len);
    }
}
```

从上述代码可以得出柔性多态的设计思想，具体如下。

- 接口内容固定，如 IShape2 中仅定义了一个多态接口方法 dispatch()。
- 子类中重写的多态函数 dispatch 仅起到转发作用，且转发的具体函数都不是多态函数，如 Circle2 类中的 getArea()、getPerimeter()都只是普通函数，这正和普通多态接口编程思想不一致。

例如，如果现在增加一个功能：求圆内接正三角形边长。可以仅在 Circle2 类中增加一个普通方法 getTriangleLen()，再在多态方法 dispatch()中增加一个 case 开关，调用 getTriangleLen()方法就可以了。而对接口 IShape2 根本就没有修改。

从中可以看出，接口定义的多态方法 dispatch()与子类中的各普通具体方法间的关系是“间接的”，不是“直接的”，是转发关系，削弱了父子类多态方法的强关联，是实现柔性多态的关键。

3. 对 dispatch()方法参数的理解

该方法有两个参数：各具体普通函数的功能号 nID，是一个整型数，在同一模块中不能

有重复值；输入参数 in，类型是 Object，相当于泛型编程，使程序灵活，一般不要定义成具体的值或类类型。

该函数返回值是 Object 对象。若为 null，表明计算失败；若非空，则在调用方用强制类型转换才能得到所需要的结果。

一个简单测试类代码如下。

```
public class Test {
    public static void main(String []args){
        IShape2 obj = new Circle2(10.0f);
        Float result = (Float)obj.dispatch(1,null);
        System.out.println("半径 10 圆面积:"+result.floatValue());
    }
}
```

4. 进一步完善

可以看出，要完成所需功能，必须知道相应具体函数的转发 ID 号。由于 ID 号是一个整型数，表意不明显，按人的思维角度不容易记忆，按字符串记忆更符合常规习惯。因此，完善后的模块应有以下主要功能：多态模块中应该给各个具体函数赋有意义的特征字符串值；调用端通过特征字符串值查询具体函数的 ID 号；根据 ID 号执行具体的转发函数。接口定义完善如下所示。也就是说，增加了一个多态函数 query，功能是查询特征字符串对应的功能 ID 号，若没有查询到，则返回-1。

```
interface  IShape2{
    public int query(String strID);
    public Object dispatch(int nID,Object in);
}
```

以 Circle 类为例，完善后代码（仅列出不同部分）如下。

```
class Circle2 implements IShape2 {
    static Vector<String> vec = new Vector();
    static {
        vec.add("getArea");
        vec.add("getPermeter");
    }
    public int query(String strID){
        int nID = vec.indexOf(strID);
        return nID;
    }
    …… //其余略
}
```

相应的测试类代码如下所示。

```
public class Test {
    public static void main(String []args){
    Shape obj = new Circle2(10.0f);
    int nID = obj.query("getArea");
    Float result = (Float)obj.dispatch(nID,null);
    System.out.println("半径 10 圆面积:"+result.floatValue());
    }
}
```

5. 总结

固化父类接口函数定义，子类通过重写多态派发函数，是柔性多态的基本设计思想。

本例中 IShape2 接口是各种形状子类的父类，其实它还可以做其他任何需要柔性多态模块的共同父类。因此接口名可取得更一般些，例如接口定义如下。

```
interface Flexible_Interface{
    public int query(String strID);
    public Object dispatch(int nID,Object in);
}
```

1.3.2　借鉴 STL 标准模板库

1. 基本思路

STL（Standard Template　Library）是标准模板库的简称，属于 C++知识体系。与 Java 语言相比，STL 的优势是它有 100 个左右的泛型方法，涵盖了绝大多数常用算法，而 JDK 中仅有全排序、二分查找等少量算法。因此，把 STL 中的泛型方法代码移植到 Java 中是一个较好的开发思路。

STL 能实现泛型的一个重要因素是 C++支持操作符重载，主要是 operator==、operator< 两个二元操作符。Java 语言可以用接口来替换，假设为 IComparator 比较器接口，为了使用方便，抽象类 AbstractComparator 定义了该接口的一个默认实现。代码如下。

```
//IComparator.java
public interface IComparator<T> {
    boolean equal(T x, T y);
    boolean less(T x, T y);
}

//AbstractComparator.java
public class AbstractComparator<T> implements IComparator<T> {
    public boolean equal(T x, T y) {
        return true;
    }
    public boolean less(T x, T y) {
        return true;
    }
}
```

2. 典型示例

【例 1-3】 编制三个对象求中值的算法。

为了说明问题，列出了采自 VC 6.0 的 STL 中该算法源码，具体如下。

```
template<class _Ty> inline
_Ty _Median(_Ty _X, _Ty _Y, _Ty _Z){
if (_X < _Y)
        return (_Y < _Z ? _Y : _X < _Z ? _Z : _X);
    else
        return (_X < _Z ? _X : _Y < _Z ? _Z : _Y);
}
```

转换后的 Java 代码如下。

```
public class Algorithm<T> {
    IComparator<T> cmp;    //比较器，是 AbstractComparator 的子类
```

```
    Algorithm (IComparator<T> cmp){
        this.cmp = cmp;   //初始化比较器
    }
    public  T median(T x, T y, T z){
        if(cmp.less(x, y))
            return cmp.less(y, z)?y:cmp.less(x, z)?z:x;
        else
            return cmp.less(x, z)?x:cmp.less(y, z)?z:y;
    }
}
```

- 由于 IComparator 接口对象为算法类 Algorithm 中所有方法共享，所以把它定义为成员变量，并在构造方法中加以初始化。
- 可以看出，C++代码与 Java 代码大同小异，主要把“<”用“less()”方法进行替换。STL 中有许多方法可以用类似这样简单的替换，而无需考虑各种细节（如边界条件等），就能直接转化成 Java 代码。得出的代码是专家级的，稳定性强。

【例 1-4】 局部排序。

STL 中排序主要包括全排序 sort()方法、局部排序 partial_sort()方法、第 nth 排序 nth_element()方法，而 JDK 中仅有全排序 sort()方法。因此，编制稳定的局部排序、求第 nth 元素排序是必要的。借鉴 STL，可以很快编制所需排序程序，以局部排序 partial_sort()为例，代码如下（在上文中的类 Algorithm 中添加）。

```
public class Algorithm<T> {
    //……略去代码同例 1-3
    public boolean push_heap(T[] t, int h, int j, T v){
        for(int i=(h-1)/2; j<h && cmp.less(t[i], v); i=(h-1)/2){
            t[h] = t[i];
            h = i;
        }
        t[h] = v;
        return true;
    }

    public boolean pop_heap(T[] t, int m, int i){
        t[i] = t[0];
        adjust_heap(t, 0, m);
        return true;
    }

    public boolean sort_heap(T[] t, int l){
        for(; l>1; --l){
            pop_heap(t, l-1, l-1);
        }
        return true;
    }

    public boolean adjust_heap(T[] t, int pos, int nSize){
        int j=pos;
        T v = t[pos];
        int k = 2*pos+2;
        for(; k<nSize; k=2*k+2 ){
```

```
            if(cmp.less(t[k], t[k-1])){
                --k;
            }
            t[pos] = t[k];
            pos = k;
        }

        if(k==nSize){
            t[pos] = t[k-1];
            pos = k-1;
        }
        push_heap(t, pos, j, v);
        return true;
    }

    public boolean make_heap(T[] t, int nSize){
        if(nSize >=2){ //保证至少有 2 个元素
            for(int i=nSize/2; i>0;){
                --i;
                adjust_heap(t, i, nSize);
            }
        }
        return true;
    }

    public boolean partial_sort(T[] t, int nSize){
        int total = t.length;
        make_heap(t, nSize); //构建堆
        for(int i=nSize; i<total; i++){
            if(cmp.less(t[i], t[0])){
                pop_heap(t, nSize, i);
            }
        }
        sort_heap(t, nSize);
        return true;
    }
}
```

第2章 反　　射

2.1 反射的概念

Java 反射（Java Reflection）是指在程序运行时获取已知名称的类或已有对象的相关信息的一种机制，包括类的方法、属性、父类等信息，还包括示例的创建和示例类型的判断等。在常规程序设计中，我们调用类对象及相关方法都是显示调用的。例如

```
public class A{
    void func() { }
    public static void main(String []args){
        A obj = new A();
        obj.func();
    }
}
```

那么能否根据类名 A，①列出这个类有哪些属性和方法，②对于任意一个对象，调用它的任意一个方法？这也是“反过来映射—反射”的含义。

在 JDK 中，主要由以下类来实现 Java 反射机制，这些类都位于 java.lang.reflect 包中。

- Class 类：代表一个类。
- Constructor 类：代表类的构造方法。
- Field 类：代表类的成员变量（成员变量也称为类的属性）。
- Method 类：代表类的方法。

2.2 统一形式调用

运用上述 Class、Constructor、Field、Method 四个类，能实现解析无穷多的系统类和自定义类结构，创建对象及方法执行等功能，而且形式是一致的。

【例 2-1】 统一形式解析类的构造方法、成员变量、成员方法。

```
import java.lang.reflect.*;
public class A {
    int m;
    public A(){}
    public A(int m){ }
    private void func1(){
```

```
}
public void func2(){
}

public static void main(String []args) throws Exception{
     // 加载并初始化指定的类 A
    Class classInfo = Class.forName("A");//代表类名是 A

    //获得类的构造函数
    System.out.println("类 A 构造函数如下所示:");
    Constructor cons[] = classInfo.getConstructors();
    for(int i = 0; i < cons.length; i++)
          System.out.println(cons[i].toString());

    //获得类的所有变量
    System.out.println();
    System.out.println("类 A 变量如下所示:");
    Field fields[] = classInfo.getDeclaredFields();
    for(int i = 0; i < fields.length; i++)
          System.out.println(fields[i].toString());

    //获得类的所有方法
    System.out.println();
    System.out.println("类 A 方法如下所示: ");
    Method methods[] = classInfo.getDeclaredMethods();
    for(int i = 0; i < methods.length; i++)
          System.out.println(methods[i].toString());
    }
}
```

- A 是自定义类。首先通过静态方法 Class.forName("A")返回包含 A 类结构信息的 Class 对象 classInfo,然后通过 Class 类中的 getConstructors()方法获得 A 类的构造方法信息，通过 getDeclaredFields()方法获得类 A 的成员变量信息，通过 getDeclaredMethods()方法获得类 A 的成员方法信息。获得其他类的结构信息步骤与上述是相似的，因此形式是统一的。
- 上述程序仅解析了 A 类的结构，并没有产生类 A 的示例。怎样用反射机制产生类 A 的示例呢？请参考下面的示例。

【例 2-2】 统一形式调用构造方法示例。

```
import java.lang.reflect.*;
public class A {
public A(){
    System.out.println("This is A:");
}
public A(Integer m){
    System.out.println("this is "+ m);
}
public A(String s, Integer m){
    System.out.println(s + ":"+m);
}
public static void main(String []args) throws Exception{
    Class classInfo = Class.forName("A");
```

```
        //第 1 种方法
        Constructor cons[] = classInfo.getConstructors();

        //调用无参构造函数
        cons[2].newInstance();
        //调用 1 个参数构造函数
        cons[1].newInstance(new Object[]{10});
        //调用 2 个参数构造函数
        cons[0].newInstance(new Object[]{"Hello",2010});

        //第 2 种方法
        System.out.println("\n\n\n");
        //调用无参构造函数
        Constructor c = classInfo.getConstructor();
        c.newInstance();

        //调用 1 个参数构造方法
        c = classInfo.getConstructor(new Class[]{Integer.class});
        c.newInstance(new Object[]{10});

        //调用 2 个参数构造方法
        c = classInfo.getConstructor(new Class[]{String.class, Integer.class});
        c.newInstance(new Object[]{"Hello", 2010});
    }
}
```

- 可以看出，反射机制有两种生成对象示例的方法。一种是通过 Class 类的无参 getConstructors()方法，获得 Constructor 对象数组，其长度等于反射类中实际构造方法的个数。示例中，cons[2]、cons[1]和 cons[0]分别对应无参、单参数、双参数 3 个构造方法，分别调用 newInstance()方法，才真正完成 3 个示例的创建过程。一种是通过 Class 类的有参 getConstructor()方法，来获得对应的一个构造方法信息，然后调用 newInstance()方法，完成该示例的创建过程。
- 加深对 Class 类中 getConstructor()方法参数的理解，其原型定义如下所示。

```
public Constructor<T> getConstructor(Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException {
```

parameterTypes 表示必须指明构造方法的参数类型，可用 Class 参数数组或依次输入参数形式来表示传入参数类型。

如果示例中要产生 A(String s, Integer m)构造方法的示例，由于第 1 个参数是字符串，第 2 个参数类型是整型数的包装类。若依次传入参数类型，则如下所示。

```
c = classInfo.getConstructor(String.class, Integer.class);
```

若传入的是数组类型，则如下所示。

```
c = classInfo.getConstructor(new Class[]{String.class, Integer.class});
```

- 加深对 Constructor 类中 newInstance()方法参数的理解，其原型定义如下所示。

```
public T newInstance(Object ... initargs)
    throws InstantiationException, IllegalAccessException,
        IllegalArgumentException, InvocationTargetException
```

initargs 表示必须指明构造方法的参数值（非参数类型），可用 Object 数组或依

次输入形式来表示传入参数值。如示例中要产生 A(String s, Integer m)构造方法的示例，由于第 1 个参数是字符串数值，第 2 个参数是整型数值。若依次传入参数类型，则如下所示。

```
c.newInstance("Hello", 2010);
```

若传入的是数组值，则如下所示。

```
c.newInstance(new Object[]{"Hello", 2010});
```

- 通过文中两种创建示例的方法对比，第 2 种方法更好，即先用有参 getConstructor()方法获得构造方法的信息，再用有参 newInstance()方法产生类的示例。

需要注意的是，在构造方法的第一次调用过程中，构造方法编译时类似栈结构，先进后出。所以 A()最先进栈，对应数组元素 cons[2]；A(Integer m)次之进栈，对应数组元素 cons[1]；A(String s, Integer m)最后进栈，对应数组元素 cons[0]。

【例 2-3】 统一形式调用成员方法示例。

```
import java.lang.reflect.*;
public class A {
    public void func1(){
        System.out.println("This is func1: ");
    }
    public void func2(Integer m){
        System.out.println("This is func2: "+m);
    }
    public void func3(String s, Integer m){
        System.out.println("This is func3: "+s+m);
    }
public static void main(String []args) throws Exception{
    Class classInfo = Class.forName("A");

    //调用无参构造函数，生成新的示例对象
    Object obj = classInfo.getConstructor().newInstance();

    //调用无参成员函数 func1
    Method mt1 = classInfo.getMethod("func1");
    mt1.invoke(obj);

    //调用 1 个参数成员函数 func2
    Method mt2 = classInfo.getMethod("func2", Integer.class);
    mt2.invoke(obj, new Object[]{10});

    //调用 2 个参数成员函数 func3
    Method mt3 = classInfo.getMethod("func3", String.class, Integer.class);
        mt3.invoke(obj, new Object[]{"Hello", 2010});
    }
}
```

方法反射主要是利用 Class 类的 getMethod()方法，得到 Method 对象，然后利用 Method 类中的 invoke()方法完成反射方法的执行。getMethod()及 invoke()方法原型及使用方法与 getConstructor()是类似的，参见上文。

【例 2-4】 一个通用方法。

分析：只要知道类名字符串、方法名字符串、方法参数值，运用反射机制就能执行该方法，程序代码如下。

```
boolean Process(String className, String funcName, Object[] para) throws Exception{
    //获取类信息对象
    Class classType = Class.forName(className);
    //形成函数参数序列
    Class c[] = new Class[para.length];
    for(int i=0; i<para.length; i++){
        c[i] = para[i].getClass();
    }

    //调用无参构造函数
    Constructor ct = classType.getConstructor();
    Object obj = ct.newInstance();
    //获得函数方法信息
    Method mt = classType.getMethod(funcName, c);
    //执行该方法
    mt.invoke(obj, para);
    return true;
}
```

通过该段程序，可以看出反射机制的突出特点是：可以把类名、方法名作为字符串变量，直接对这两个字符串变量进行规范操作，就能产生类的示例及运行相应的方法，与普通的先 new 示例再直接调用所需方法有本质的不同。

2.3 反射与配置文件

2.3.1 反射与框架

反射技术的编程特点：大大提高了编制稳定框架的能力。具体表现在，当新增加功能的时候，仅增加相应的功能类，而框架调用可能不需要发生变化。

【例 2-5】题目同例 1-2：求圆和长方形的面积。要求：从命令行输入类名字符串。当输入“Circle”时,表明求圆的面积；当输入“Rect”时，表明求矩形的面积。

```
//IShape.java : 形状接口定义
public interface IShape {                    //同例 1-2
    boolean input();                         //输入方法
    float getArea();                         //求面积方法
}
//ShapeProc.java: 流程处理类
public class ShapeProc {                     //同例 1-2
    private IShape shape;
    public ShapeProc(IShape shape){
        this.shape = shape;
    }

    public float process(){                  //每个形状处理包括输入及求面积两步
        shape.input();                       //输入功能
```

```
            float value = shape.getArea(); //求面积功能
            return value;           //返回面积
        }
}

public class Circle implements Shape {
    public Circle() {}
    //略，代码同例 1-2
}
public class Rect implements Shape {
    public  Rect(){ }
    //略，代码同例 1-2
}

//仅以下测试类不同
public class Test {
    public static void main(String []args)throws Exception{
        IShape shape = null;
        shape = (IShape)Class.forName(args[0]).
getConstructor().newInstance();
        ShapeProc Obj = new ShapeProc(shape);
        float value = Obj.process();
        System.out.println("所求面积是:" + value);
    }
}
```

- 反射机制要求待反射的类中的构造方法必须显示化，即使是无参默认构造方法，也要显示地写出来。
- 着重加深对测试类代码的理解。命令行参数表示要求类名为 args[0]形状的面积。其中最重要的一行代码如下所示。

```
Class.forName(args[0]).getConstructor().newInstance()
```

它表明产生了类名是 args[0]的一个示例。可能是 Circle，可能是 Rect，也可能是其他形状的一个示例。也就是说，该行表明的含义是动态的，不随哪个具体的 IShape 接口的子类改变而改变。从框架角度来说，它是稳定的。例如，现在要增加一个求三角形面积的类，只需增加功能子类 Triangle，而测试类代码无需改变。

- 本例中，产生形状对象的示例用了反射技术，调用方法时，并没有用到反射技术，而是用到了接口技术。反射技术固然强大，但它是以牺牲时间为代价的，代码也不易理解。一般来说，运用“接口+构造方法反射”就足以编制功能强大的框架代码了。
- 因此，如果把某些动态参数封装在配置文件中，通过读取配置文件获得所需参数，再运用反射技术，就可以编制更加灵活的代码，是下文即将论述的内容。

2.3.2 Properties 配置文件

Properties 格式文件是 Java 常用的配置文件，是简单的文本格式。它是用来在一个文件中存储键-值对的，其中键和值用等号分隔。JDK 中利用系统类 Properties 来解析 Properties 文件。Properties 类是 Hashtable 的一个子类，用于键 keys 和值 values 之

间的映射。Properties 类表示一个持久的属性集，属性列表中每个键及其键值都是一个字符串。其常用函数如下。

- Properties()：创建一个无默认值的空属性列表。
- void load(InputString inStream)：从输入流中读取属性列表。
- String getProperty(String key)：获取指定键 key 的键值，以字符串的形式返回。
- void setProperty(String key, String value)：设置对象中 key 的键值。

【例 2-6】求圆和长方形面积。要求定义 properties 文本文件 shape.properties,其中添加一个键值对：shape=Circle。

```
public interface IShape {……}                          //同例 2-5
public class ShapeProc {……}                            //同例 2-5
public class Circle implements IShape {……}  //同例 2-5
public class Rect implements IShape {……}     //同例 2-5

//仅以下测试类不同
public class Test{
    public static void main(String []args)throws Exception{
        Properties p = new Properties();
        p.load(new FileInputStream("d:/shape.properties")); //装载配置文件
        //根据键"shape",获取类名字符串
        String cname = p.getProperty("shape");
        IShape shape = null;
        shape = (IShape)Class.forName(cname).
                getConstructor().newInstance();
        ShapeProc Obj = new ShapeProc(shape);
        float value = Obj.process();
        System.out.println("所求面积是:" + value);
    }
}
```

Properties 还支持如表 2-1 所示的简单 XML 格式文件（重新定义 shape.properties 文件为 shape.xml）。

表 2-1　　shape.xml 定义

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>求圆和三角形面积</comment>
    <entry key=shape>Circle</entry>
</properties>
```

其 XML 文件要求如下：一个 properties 标签，一个 comment 注释子标签，然后是任意数量的〈entry〉标签。对每一个〈entry〉标签，有一个键属性，输入的内容就是它的值。利用 Properties 类解析该文件与解析老式文本文件几乎是一致的，只不过用 loadFromXML()代替了 load()方法。代码如下。

```
//仅测试类与例 2-6 稍有不同
public class Test {
    public static void main(String []args)throws Exception{
```

```
Properties p = new Properties();
//用 loadFromXML()代替了 load()方法
p.loadFromXML(new FileInputStream("d:/shape.properties"));
String cname = p.getProperty("shape"); //根据键"shape", 获取类名字符串
IShape shape = null;
shape = (IShape)Class.forName(cname).getConstructor().newInstance();
ShapeProc Obj = new ShapeProc(shape);
float value = Obj.process();
System.out.println("所求面积是:" + value);
```

第3章 工厂模式

3.1 问题的提出

众所周知，在现实生活中，工厂是用来生产产品的，有两个关键的角色：产品及工厂。计算机中的工厂模式与实际工厂的特征是相近的，因此工厂模式的关键点就是如何描述好这两个角色之间的关系，分为四种情况，具体如下。

（1）单一产品系。工厂生产一种类型的产品。表 3-1 表明是小汽车工厂，生产高、中、低档 3 种类型小汽车。

表 3-1　　小汽车工厂

名称 \ 种类	高档	中档	低档
小汽车	√	√	√

（2）多产品系，特征相同。工厂生产多种类型的产品。表 3-2 表明是小汽车、公共汽车工厂，均有高、中、低档 3 种类型的汽车。

表 3-2　　小汽车、公共汽车工厂

名称 \ 种类	高档	中档	低档
小汽车	√	√	√
公共汽车	√	√	√

（3）多产品系，部分特征相同。表 3-3 表明是小汽车、公共汽车工厂，均有高档、中档两种类型的汽车。除此之外，小汽车还有低档类型。

表 3-3　　小汽车、公共汽车工厂

名称 \ 种类	高档	中档	低档
小汽车	√	√	√
公共汽车	√	√	

（4）多产品系，无特征相同。表 3-4 表明是小汽车、公共汽车工厂，小汽车有高档、中档两种类型，公共汽车仅有低档类型。

表 3-4　　小汽车、公共汽车工厂

种类 / 名称	高档	中档	低档
小汽车	√	√	
公共汽车			√

工厂模式一般分为简单工厂、工厂、抽象工厂 3 种情况，属于创建型设计模式。上面的工厂生产汽车的几种情况可以分别映射到这几种工厂模式中，下面分别加以描述。

3.2 简 单 工 厂

3.2.1 代码示例

【例 3-1】 编制表 3-1 小汽车简单工厂模式的相关类。

其对应的 UML 框图如图 3-1 所示。

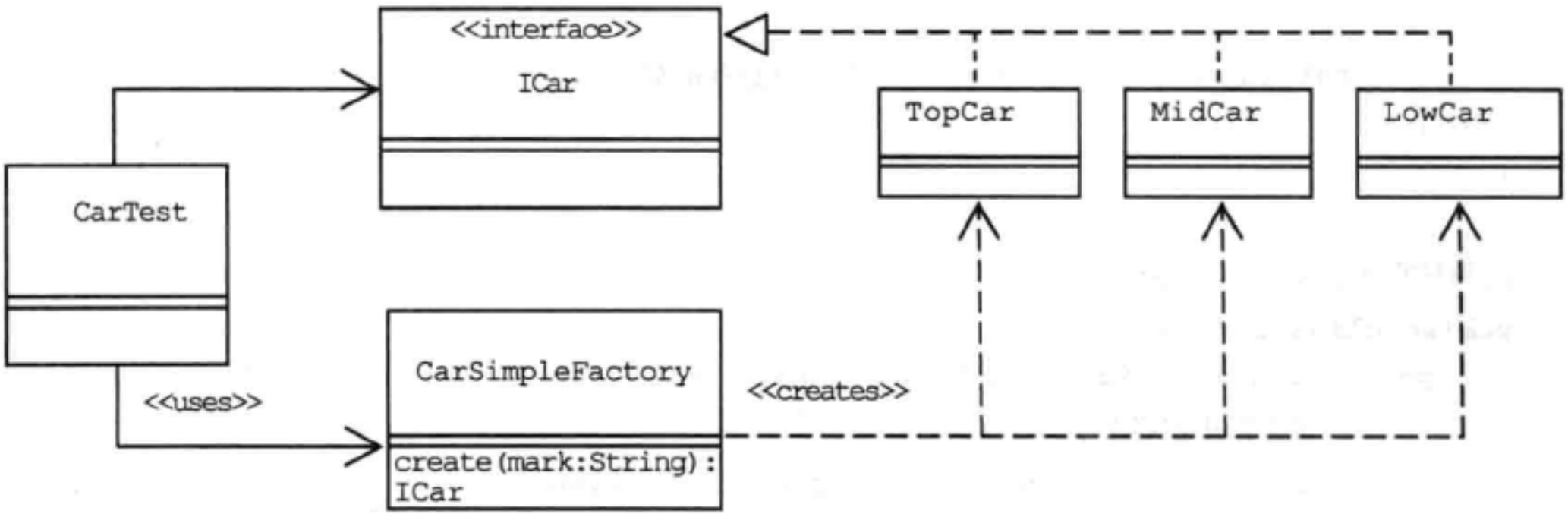


图 3-1　小汽车简单工厂 UML 示意图

```
//定义小汽车接口：ICar.java
public interface ICar {
    //由于工厂模式仅关系对象的创建，为说明方便，无需定义方法
}

//下面定义高、中、低档具体的小汽车
//高档小汽车：TopCar.java
public class TopCar implements ICar {
}
```

```
//中档小汽车: MidCar.java
public class MidCar implements ICar {
}

//低档小汽车: LowCar.java
public class LowCar implements ICar {
}

//简单工厂:CarSimpleFactory.java
public class CarSimpleFactory {
    public static final String TOPTYPE="toptype";
    public static final String MIDTYPE = "midtype";
    public static final String LOWTYPE = "lowtype";

    public static ICar create(String mark){
        ICar obj = null;
        if(mark.equals(TOPTYPE)){        //如果是高档类型
            obj = new TopCar();          //则创建高档车对象
        }
        else if(mark.equals(MIDTYPE)){
            obj = new MidCar();
        }
        else if(mark.equals(LOWTYPE)){
            obj = new LowCar();
        }
        return obj;                      //返回选择的对象
    }
}

//测试程序:CarTest.java
public class CarTest {
    public static void main(String[] args) {
        //从工厂中创建对象
        ICar obj = CarSimpleFactory.create("toptype");
    }
}
```

3.2.2 代码分析

1. 简单工厂功能类编制步骤

- 定制抽象产品接口，如 ICar。
- 定制具体产品子类，如类 TopCar、MidCar、LowCar。
- 定制工厂类，如 CarSimpleFactory。简单工厂类的特点：它是一个具体的类，非接口或抽象类。其中有一个重要的 create()方法，利用 if…else 或 switch 开关创建所需产品，并返回。

2. 工厂类静态 create( )方法的理解

使用简单工厂的时候，通常不用创建简单工厂类的类示例，没有创建示例的必要。因此可以把简单工厂类实现成一个工具类，直接使用静态方法就可以了。也就是说，简单工厂的方法通常是静态的，所以也被称为静态工厂。如果要防止客户端无谓地创造简单工厂示例，还可以把简单工厂的构造方法私有化。

3.2.3 语义分析

3.2.2 小节中的代码分析是绝大多数设计模式书中论述过的内容。那么，能否有更简明的方法加以说明呢？很明显，生活中的语义分析是一个优秀的方法，可以方便构建应用程序的框架。例如甲和乙关于工厂和工作的对话，如下所示。

甲：你在哪里上班?

乙：小汽车工厂。

甲：做什么工作?

乙：生产小汽车。

甲：生产几种小汽车?

乙：高、中、低档 3 种小汽车。

再如甲和乙关于旅游的对话，如下所示。

甲：去过北京吗?

乙：去过。

甲：北京哪里好玩?

乙：长城。

甲：长城哪里最好玩?

乙：八达岭。

再如，所有书籍一般都有书名，一、二、三级标题等。可以得出生活中语义描述事物的一个显著特点：范围从大到小，从泛泛到具体，从一般到特殊。因此计算机程序结构一定也要遵循这种结构，即按层次划分。本示例简单工厂模式语义划分层次描述如表 3-5 所示。

表 3-5 示例简单工厂模式语义层次划分

产品角色语义分层：
① 小汽车产品： 泛指，与接口 ICar 对应。
② 高、中、低档三类小汽车，与 UpCar、MidCar、DnCar 一一对应。
工厂角色语义分层：
① 工厂可管理三类小汽车。
② 在工厂类中可直接看出创建产品种类的数目，是简单工厂的最大特点。

如果现在又新增了一个超高档类型的汽车，在简单工厂模式下，需要做的工作有：①新增 ICar 的子类 SuperCar；②修改工厂类 SimpleCarFactory 中的 create()方法，增加 SuperCar 对象的判断选择分支。类 SuperCar 的增加是必然的，那么能否不修改工厂类就能完成所需功能呢？这就是下面要论述的工厂模式。

3.3 工　厂

3.3.1 代码示例

【例 3-2】 编制表 3-1 所示小汽车工厂模式的相关类。

其对应的 UML 框图如图 3-2 所示。

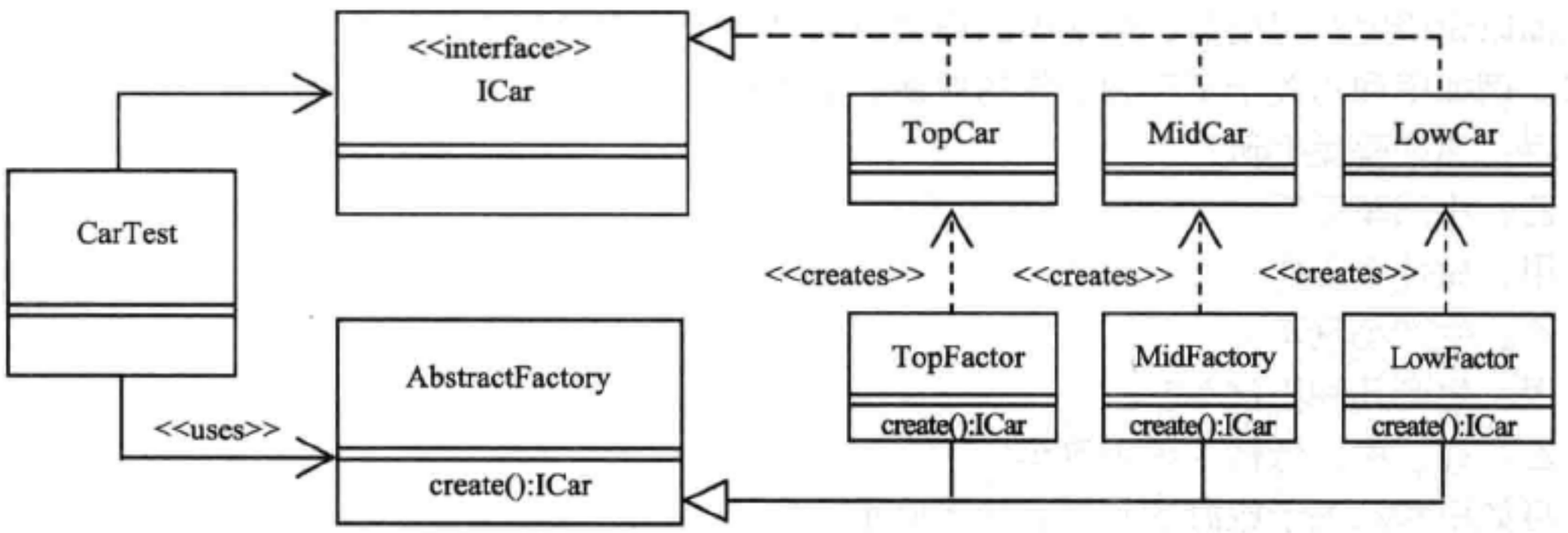


图 3-2　小汽车抽象工厂 UML 示意图

```
//定义小汽车接口：ICar.java
public interface ICar {
    //由于工厂模式仅关系对象的创建，为说明方便，无需定义方法
}

//定义高、中、低档具体的小汽车
//高档小汽车：TopCar.java
public class TopCar implements ICar {
}

//中档小汽车：MidCar.java
public class MidCar implements ICar {
}

//低档小汽车：LowCar.java
public class LowCar implements ICar {
}

//定义抽象工厂：AbstractFactory.java
public abstract class AbstractFactory {
    public abstract ICar create();
}

//定义高档小汽车工厂：TopFactory.java
public class TopFactory extends AbstractFactory {
    public ICar create() {
```

```
            return new TopCar();  //高档工厂生成高档小汽车对象
        }
}

//定义中档小汽车工厂：MidFactory.java
public class MidFactory extends AbstractFactory {
    public ICar create() {
            return new MidCar();  //中档工厂生成中档小汽车对象
    }
}

//定义低档小汽车工厂：LowFactory.java
public class LowFactory extends AbstractFactory {
    public ICar create() {
            return new LowCar();  //低档工厂生成低档小汽车对象
    }
}

//测试类：CarTest.java
public class CarTest {
    public static void main(String []args){
            AbstractFactory obj = new TopFactory();//多态创建高档工厂
            ICar car = obj.create();                    //获得高档工厂中的小汽车对象
    }
}
```

3.3.2 代码分析

1. 工厂模式功能类编制步骤

- 定制抽象产品接口，如 ICar。
- 定制具体产品子类，如类 TopCar、MidCar、LowCar。
- 定制抽象工厂类（或接口），如 AbstractFactory。其中有一个重要的 create()抽象方法。
- 定制具体工厂子类，如 UpFactory、MidFactory、DnFactory。

2. 工厂与简单工厂模式的区别

- 工厂模式把简单工厂中具体的工厂类（如 CarSimpleFactory）划分成两层：抽象工厂层（如 AbstractFactory）+具体工厂子类层（如 TopFactory 等）。抽象工厂层的划分丰富了程序框架的内涵，符合从一般到特殊的语义特点。以本题为例，语义的详细描述如表 3-6 所示。

表 3-6　　抽象产品层+抽象工厂层语义描述

抽 象 层	语 义 描 述
抽象产品层 `interface ICar{ …… }`	生产小汽车产品，小汽车的种类是不确定的

抽象工厂层
`abstract class AbstractFactory{`
`  public abstract ICar create();`
`}`
	工厂管理小汽车产品，管理小汽车的种类是不确定的

其实，表中代码最简单的语义描述就是“我们是小汽车工厂，生产并管理小汽车”。也就是说，抽象产品+抽象工厂定义好了，需要完成的工作基本也就清晰了，就能转化成有意义的语义描述。事实上，只要有 ICar.java、AbstractFactory.java 两个文件，其他具体的产品类、工厂类源文件都没有，编译仍能通过。这进一步证明了生活中“一般到特殊”的特点，在程序设计中也一定而且是必然存在的。因此，把程序中的代码用生活中的语言描述出来，看是否符合实际，是一个非常好的习惯。

- create()方法参数的理解：在简单工厂中，create(String mark)是成员方法，表明在该方法中管理多个产品，根据 mark 的值产生并返回 ICar 对象；在工厂模式中，create()是抽象方法,无参数，表明在具体的子类工厂中创建某个具体的产品。
- 工厂方法更易于软件的二次开发及维护，主要特征是：当需求分析发生变化时，只需要增加、删除相应的类，而不是修改已有的类。例如，若又生产一种超高档的小汽车，只需要增加 SuperCar 及 SuperFactory 两个类即可，代码如下。

```
//超高档小汽车：SuperCar.java
public class SuperCar implements ICar {
}
//定义超高档小汽车工厂：SuperFactory.java
public class SuperFactory extends AbstractFactory {
    public ICar create() {
        return new SuperCar();//超高档工厂生成超高档小汽车对象
    }
}
```

若在简单工厂中，则必须修改 CarSimpleFactory 工厂类中的 create()方法，增加选择分支。因此可以看出，工厂模式优于简单工厂模式。

3.4 抽 象 工 厂

一般来说，简单工厂、工厂模式是单产品系的，抽象工厂是多产品系的。从本质上来说，抽象工厂、工厂模式是统一的。

3.4.1 代码示例

【例 3-3】 编制表 3-2 所示汽车抽象工厂模式的相关类。

```
//以下小汽车接口，高、中、低档小汽车代码与例 3-2 相同
public interface ICar {  }
public class TopCar implements ICar {  }
public class MidCar implements ICar {  }
public class LowCar implements ICar {  }

//定义公共汽车接口、高、中、低档公共汽车类
public interface IBus {  }
public class UpBus implements IBus {  }
public class MidBus implements IBus {  }
public class DnBus implements IBus {  }
```

```
//定义抽象工厂：AbstractFactory.java
public absttract class AbstractFactory {
    public abstract ICar createCar();  //产生小汽车对象
    public abstract IBus createBus();  //产生公共汽车对象
}

//定义高档工厂：TopFactory.java
public class TopFactory extends AbstractFactory {
    public ICar createCar() {
        return new TopCar();        //高档工厂生成高档小汽车对象
    }
    public IBus createBus() {
        return new UpBus();         //高档工厂生成高档公共汽车对象
    }
}

//定义中档工厂：MidFactory.java
public class MidFactory extends AbstractFactory {
    public ICar createCar() {
        return new MidCar();        //中档工厂生成中档小汽车对象
    }
    public IBus createBus() {
        return new MidBus();        //中档工厂生成中档公共汽车对象
    }
}

//定义低档工厂：LowFactory.java
public class LowFactory extends AbstractFactory {
    public ICar createCar() {
        return new LowCar();        //低档工厂生成中档小汽车对象
    }
    public IBus createBus() {
        return new DnBus();         //低档工厂生成中档公共汽车对象
    }
}
```

3.4.2 代码分析

（1）抽象工厂模式功能类编制步骤如下所示。

- 定制抽象产品接口，如 ICar，IBus。
- 定制具体产品子类，如小汽车类 TopCar、MidCar、LowCar，公共汽车类 UpBus、MidBus、DnBus。
- 定制抽象工厂类（或接口），如 AbstractFactory。其中有两个重要的 create()抽象方法，分别返回 ICar、IBus 对象。
- 定制具体工厂子类，如 TopFactory、MidFactory、LowFactory，每个工厂类中重写 create()方法。

（2）从本质上来说，抽象工厂与工厂模式是统一的，只不过抽象工厂是多产品系的，工厂模式是单产品系的。

3.4.3 典型模型语义分析

抽象工厂的语义描述如下：设 A 产品有 A_1，A_2，…，A_n，B 产品有 B_1，B_2，…，B_n，共享特征 C，C 有 C_1，C_2，…，C_n。即 C_1 特征的产品是 A_1、B_1，C_2 特征的产品是 A_2、B_2，…，C_n 特征的产品是 A_n，B_n。把该语义翻译成计算机程序，如表 3-7 所示。

表 3-7　　抽象工厂语义转译程序表

语　义	转译代码
① A 产品有 A_1，A_2，…，A_n	interface IA{} class A1 implements IA{} class A2 implements IA{} …… class An implements IA{}
② B 产品有 B_1，B_2，…，B_n	interface IB{} class B1 implements IB{} class B2 implements IB{} …… class Bn implements IB{}

③ 共享特征 C	abstract class C{
	　　abstract IA create();
	　　abstract IB create();
	}

④ C_1 特征产品是 A_1、B_1	class C1 extends C{
	　　IA create() { return new A1();}
	　　IB create() { return new B1();}
	}

3.4.4 其他情况

【例 3-4】 编制表 3-3 所示汽车抽象工厂模式的相关类。

分析：表 3-3 属于多产品系，局部特征情况相同，小汽车和公共汽车都有高、中档类型，小汽车有低档类型，而公共汽车没有。代码如下。

```
//以下小汽车接口，高、中、低档小汽车代码与例 3-2 相同
public interface ICar {  }

public class TopCar implements ICar {  }
public class MidCar implements ICar {  }
public class LowCar implements ICar {  }

//定义公共汽车接口，高、中档公共汽车类
public interface IBus {  }
public class UpBus implements IBus {  }
public class MidBus implements IBus {  }
```

```
//定义抽象工厂：AbstractFactory.java
public absttract class AbstractFactory {
}

//定义抽象子工厂 1：AbstractFactory1.java
public absttract class AbstractFactory1 extends AbstractFactory {
    public abstract ICar createCar();  //产生小汽车对象
    public abstract IBus createBus();  //产生公共汽车对象
}

//定义抽象子工厂 2：AbstractFactory2.java
public absttract class AbstractFactory2 extends AbstractFactory {
    public abstract ICar createCar();  //产生小汽车对象
}

//定义高档工厂：TopFactory.java
public class TopFactory extends AbstractFactory1 {
    public ICar createCar() {
        return new TopCar();              //高档工厂生成高档小汽车对象
    }
    public IBus createBus() {
        return new UpBus();               //高档工厂生成高档公共汽车对象
    }
}

//定义中档工厂：MidFactory.java
public class MidFactory extends AbstractFactory1 {
    public ICar createCar() {
        return new MidCar();              //中档工厂生成中档小汽车对象
    }
    public IBus createBus() {
        return new MidBus();              //中档工厂生成中档公共汽车对象
    }
}

//定义低档工厂：LowFactory.java
public class LowFactory extends AbstractFactory2 {
    public ICar createCar() {
        return new LowCar();              //低档工厂生成低档小汽车对象
    }
}
```

着重理解 AbstractFactory、AbstractFactory1、AbstractFactory2 的语义描述，如表 3-8 所示。

表 3-8　　三个抽象工厂类语义描述

代　码	语义描述
abstract class AbstractFactory{ }	抽象类，无方法，两个作用：①表明不同的派生子类中，功能（方法）是不同的；②对不同的功能子类进行统一管理

续表

代　码	语 义 描 述

abstract class AbstractFactory1 extends AbstractFactory{
　　public abstract ICar createCar();
　　public abstract IBus createBus();
}
	① 具有相同特征的小汽车、公共汽车放在相同的工厂中；② 该类也是抽象类，表明“特征”是多个，本例中“特征”表示“高档”及“中档”，差别要在该类的子类中体现

abstract class AbstractFactory2 extends AbstractFactory{
　　public abstract ICar createCar();
}
	对该类描述与上面基本相同

【例 3-5】 编制表 3-4 所示汽车抽象工厂模式的相关类。

分析：表 3-4 属于多产品系，无特征情况相同，小汽车有高、中档类型，小汽车无低档类型，公共汽车有低档类型。借鉴例 3-4，代码如下。

```
//定义小汽车接口及高、中档小汽车类
interface ICar {}
class TopCar implements ICar {}
class MidCar implements ICar {  }

//定义公共汽车接口，低档公共汽车类
interface IBus {}
class DnBus implements IBus {}

//定义抽象工厂：AbstractFactory.java
abstract class AbstractFactory {
}

//定义高档工厂：TopFactory.java
class TopFactory extends AbstractFactory {
    public ICar createCar() {
        return new TopCar();        //高档工厂生成高档小汽车对象
    }
}
//定义中档工厂：MidFactory.java
class MidFactory extends AbstractFactory {
    public ICar createCar() {
        return new MidCar();        //中档工厂生成中档小汽车对象
    }
}
//定义低档工厂：LowFactory.java
class LowFactory extends AbstractFactory {
    public IBus createBus() {
        return new DnBus();         //低档工厂生成中档公共汽车对象
    }
}
```

3.5 应 用 探 究

【例 3-6】 编写读文件功能。具体功能是：读取文本文件，包括（GBK,UTF8,UNICODE）

编码下的文本文件，要求获得全文内容；读取图像文件（BMP,GIF,JPG）文件，要求获得图像宽度、长度、每一点的 RGB 三基色信息。

方法 1：根据语义，分别画出图 3-3 所示功能层次图。

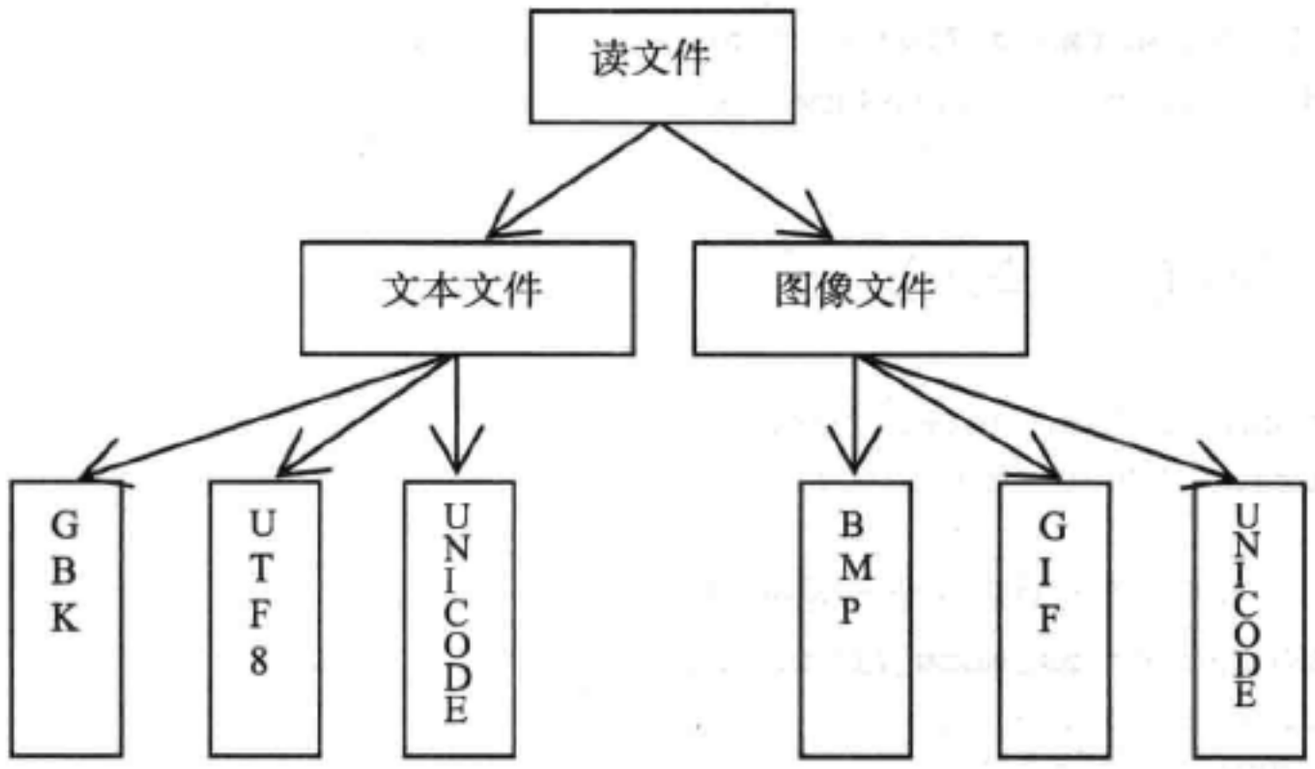


图 3-3 读文本、图像文件层次图

根据该层次图，易得出下述程序框架。

（1）定义文件产品。

```
//定义读文件接口，对应图 3-3 中第 1 层
public interface IRead {
    public void read(String fileName);
}
//定义读文本、图像文件抽象类，对应图 3-3 中第 2 层
public abstract class AbstractTextRead implements IRead {     //读文本文件
    public void read(String fileName) {
    }
}
public abstract class AbstractImgRead implements IRead {      //读图像文件
    public void read(String fileName) {
    }
}
//定义具体读文本、图像文件类，对应图 3-3 中第 3 层
public class GBKRead extends AbstractTextRead {               //GBK 编码文件
    public void read(String fileName) {
    }
}
public class UTF8Read extends AbstractTextRead {              //UTF8 编码文件
    public void read(String fileName) {
    }
}
public class UNICODERead extends AbstractTextRead {           //UNICODE 编码文件
    public void read(String fileName) {
    }
}
public class BMPRead extends AbstractImgRead {                //BMP 图像文件
    public void read(String fileName) {
    }
}
```

```
public class GIFRead extends AbstractImgRead { //GIF 图像文件
    public void read(String fileName) {
    }
}
public class JPGRead extends AbstractImgRead { //JPG 图像文件
    public void read(String fileName) {
    }
}
```

（2）定义工厂类（利用工厂模式）。

```
//定义抽象工厂类
public abstract class AbstractFactory {
    public abstract IRead create();
}
//定义具体工厂类，上述 6 个产品对应 6 个不同的工厂类
public class GBKFactory extends AbstractFactory {
    public IRead create() {
        return new GBKRead();
    }
}
public class UTF8Factory extends AbstractFactory {
    public IRead create() {
        return new UTF8Read();
    }
}
public class UNICODEFactory extends AbstractFactory {
    public IRead create() {
        return new UNICODERead();
    }
}
public class BMPFactory extends AbstractFactory {
    public IRead create() {
        return new BMPRead();
    }
}
public class GIFFactory extends AbstractFactory {
    public IRead create() {
        return new GIFRead();
    }
}
public class JPGFactory extends AbstractFactory {
    public IRead create() {
        return new JPGRead();
    }
}
```

容易发现，根据图 3-3，利用工厂模式可以直译成上述代码。但是，这种层次框架并不是最优的，关键在于没有充分运用到 JDK 本身已有的类库，没有在图 3-3 的基础上进行进一步抽象。这也就是下面方法 2 改进的地方。

方法 2：

分析：① JDK 中提供了不同编码下的字符串转换方法，其中一个重要的构造方法是：String(byte buf[], String encode)。因此得出读文本文件的思路，即按字节输入流把文件读入缓

冲区 buf，再按上述 String 构造方法，将 buf 缓冲区按 encode 编码方式进行编码，转化成可视字符串。利用这种方法不但适应 GBK、UTF8、UNICODE 编码的文本文件，还可适应其他编码文件。② JDK 中提供了图像操作类，如 ImageIO，封装了对 BMP、GIF、JPG 等格式图像文件的读写操作。利用 ImageIO，可大大减少代码编码量，提高编程效率。具体代码如下所示。

（1）定义读（文本、图像）文件接口。

读文本文件方法需要两个参数：文件名，文件编码方式；读图像文件方法需要一个参数：文件名。根据题意，读文本文件要求返回 String 类型，读图像文件要求返回图像长、宽、RGB 复合信息。如何用接口屏蔽方法参数个数，返回值类型的差异，是定义接口的关键。本文定义的接口如下所示。

```
public interface IRead<T>{
    T read(String ... in);
}
```

定义泛型接口是解决返回值类型不同的较好方法，而屏蔽方法参数个数差异利用“String … in”形式即可实现。根据该接口，利用工厂模式，得出 UML 如图 3-4 所示。

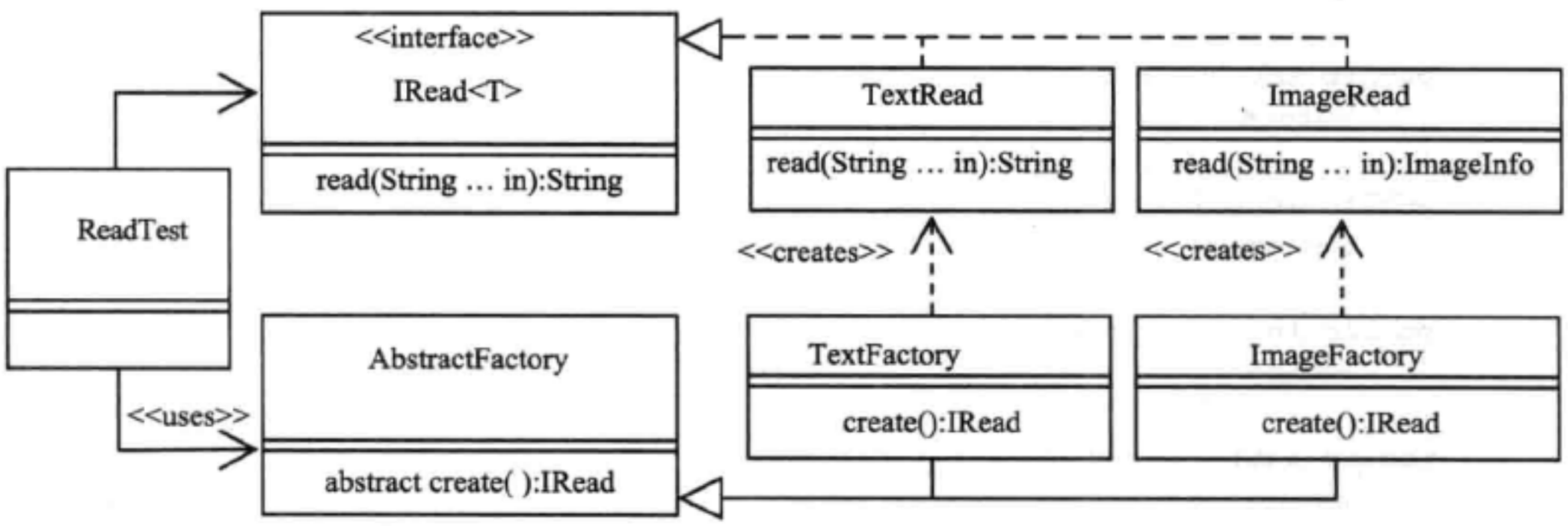


图 3-4 读文件 UML 结构图

（2）定义读文本文件、图像文件具体类。

```
//读文本文件
public class TextRead implements IRead<String> {     //读文本文件
    public String read(String... in) {                //可输入 0 或多个参数
        String result = null;                         //result 是结果串
        try{
            File file = new File(in[0]);              //in[0]表示文件名称
            long len = file.length();
            FileInputStream input = new FileInputStream(in[0]);
            byte buf[] = new byte[(int)len];          //缓冲区大小等于文件长度
            input.read(buf);                          //一次读完文件
            result = new String(buf, in[1]);          //按 in[1]编码方式转化成可见字符串
            input.close();
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
```

```
            return result;
        }
    }

    //图像基本信息文件
    public class ImageInfo {
        private int width;          //图像宽度
        private int height;         //图像高度
        private int r[][];          //红色分量
        private int g[][];          //绿色分量
        private int b[][];          //蓝色分量

        public void setWidth(int width){
            this.width = width;
        }
        public int getWidth() {
            return width;
        }
        public void setHeight(int height){
            this.height = height;
        }
        public int getHeight() {
            return height;
        }
        public int[][] getR() {
            return r;
        }
        public int[][] getG() {
            return g;
        }
        public int[][] getB() {
            return b;
        }

        public void setRGB(int rgb[]){
            r = new int[height][width];
            g = new int[height][width];
            b = new int[height][width];

            int pos = 0;
            for(int i=0; i<height; i++){
                pos = width*i;
                for(int j=0; j<width; j++){
                    r[i][j] =(rgb[pos+j]&0xff0000)>>16;
                    g[i][j] =(rgb[pos+j]&0x00ff00)>>8;
                    b[i][j] =rgb[pos+j]&0x0000ff;
                }
            }
        }
    }

    //读图像文件
    public class ImageRead implements IRead<ImageInfo> { //读图像文件
```

```
public ImageInfo read(String... in) {
    BufferedImage bi=null;
    File f = new File(in[0]);                          //in[0]表示图像文件名
    try {
        bi = ImageIO.read(f);
    } catch (IOException e) {
        e.printStackTrace();
    }
    int width = bi.getWidth();
    int height = bi.getHeight();
    int rgb[] = new int[width*height];
    //将图像数据读到 result 缓冲区
    bi.getRGB(0, 0, width, height, rgb, width, height);
    ImageInfo obj = new ImageInfo();                   //设置图像信息
    obj.setWidth(width);                               //设置宽度
    obj.setHeight(height);                             //设置高度
    obj.setRGB(rgb);                                   //设置 rgb[]三基色信息
    return obj;
  }
}
```

- 自定义 ImageInfo 类与 JDK 系统类 ImageIO 完成了对图像文件的读操作。ImageIO 中的 read()方法可把某图像矩形区域像素点的三基色值统一读到一维整型数组中，因此必须对该数组每一值进行移位拆分，获得分别的 r、g、b 值。拆分过程如 ImageInfo 中的 setRGB()方法所示。
- 利用 IMageIO 类，简化了读不同格式图像文件基础类的编制，仅有一个类就可以了。ImageIO 类不但有读功能，还有写功能，那么都支持哪些图像文件呢？执行下列代码就可看出来。

```
String readSuffix[] = ImageIO.getReaderFileSuffixes();//获得可读图像文件类型扩展名
for(int i=0; i<readSuffix.length; i++){
    System.out.println(readSuffix[i]);
}
String writeSuffix[] = ImageIO.getWriterFileSuffixes();//获得可写图像文件类型扩展名
for(int i=0; i<writeSuffix.length; i++){
    System.out.println(writeSuffix[i]);
}
```

（3）定义抽象工厂。

```
public abstract class AbstractFactory {
    public abstract IRead create();
}
```

（4）定义具体工厂。

```
public class TextFactory extends AbstractFactory {
    public IRead create() {
        return new TextRead();     //文件工厂产生具体读文件类
    }
}
public class ImageFactory extends AbstractFactory {
    public IRead create() {
```

```
            return new ImageRead();    //图像工厂产生具体读图像类
        }
    }
```

3.6 自动选择工厂

例 3-6 给出了工厂类的功能代码，如何选择具体工厂类并没有体现。其实，与简单工厂类中选择分支一样，在抽象类中加相应的代码就可以了，如下所示。

```
public abstract class AbstractFactory {
    private static String TEXT = "text";
    private static String IMAGE= "IMAGE";

    public abstract IRead create();

    static AbstractFactory create(String mark){  //是具体工厂类型标识字符串，不是类名
        AbstractFactory factory = null;
        if(mark.equals(TEXT))
            factory = new TextFactory();
        if(mark.equals(IMAGE))
            factory = new ImageFactory();
        return factory;
    }
}
```

抽象方法 create()语义是：具体工厂类对象是由客户端调用方产生的；静态方法 create()语义是：具体工厂类对象是在本类产生的，根据 mark 标识自动产生不同的具体工厂类对象。本类暗含了两种产生工厂对象的方法，方便用户加以选择。

其实，运用 Java 反射技术，可以编制更加灵活的代码，如下所示。

```
import java.lang.reflect.*;
public abstract class AbstractFactory {
    public abstract IRead create();

    static AbstractFactory create(String className){ //className 是具体的工厂类名字符串
        AbstractFactory factory = null;
        try{
            Class c = Class.forName(className);
            factory = (AbstractFactory)c.newInstance();
        }
        catch(Exception e){
            e.printStackTrace();
        }
        return factory;
    }
}
```

运用反射技术，实现了更加灵活的自动工厂选择功能。当增加新具体工厂类的时候，无需修改 AbstractFactory 类。仔细分析，该类结构对抽象工厂模式是最恰当的。抽象工厂对应多产品簇，每个具体工厂包含多种产品。从层次清晰角度来说，也应该先得到具体工厂，再得到该工厂中的某个具体产品。但是对于简单工厂、工厂模式而言，它们都对应单一产品簇。在运用反射技术的前提下，没有必要利用反射先产生具体工厂，再产生具体产品，直接用反

射产生具体产品就可以了。而且该类也可由抽象类变成普通类，代码如下。

```
public class ProductFactory {
    static IRead create(String className){ //className 是某具体产品类名，非工厂类名
        IRead product= null;
        try{
            Class c = Class.forName(className);
            product = (IRead)c.newInstance();
        }
        catch(Exception e){
            e.printStackTrace();
        }
        return product;
    }
}
```

分析代码得出，产品工厂类 ProductFactory 是用于返回 IRead 产品的。稍加改造，运用泛型技术，可以得出更一般的形式，代码如下。

```
public class ProductFactory2<T> {                    //T: 定义的产品接口
    public T create(String className){               //className: 具体的产品类名
        T product= null;                             //产品初始化为 null
        try{
            Class c = Class.forName(className);
            product = (T)c.newInstance(); //强制转换
        }
        catch(Exception e){
            e.printStackTrace();
        }
        return product;
    }
}
```

第4章　生成器模式

4.1　问题的提出

在类的应用中，有些类是容易创建对象的，直接调用构造方法即可。例如，Student obj = new Student("1001", "张三",20),表明学生学号是 1001,姓名是张三,年龄 20;Circle obj2 = new Circle(10.0f)，表明创建一个半径是 10 的圆对象。这两个类的一大特点是成员变量是基本数据类型或其封装类，或是字符串类。有些类是不易直接创建对象的，成员变量是自定义类型，代码如下。

```
public class Product {
    Unit1 u1;
    Unit2 u2;
    Unit3 u3;
}
```

可以看出，Product 由 Unit1、Unit2、Unit3 三个单元组成，产生 Product 对象不能简单地由 Product obj=new Product(Unit1,Unit2,Unit3)获得，必须先产生具体的对象 u1、u2、u3，然后才能获得 Product 对象，简单的实现方法如下。

```
public class Product {
    Unit1 u1;
    Unit2 u2;
    Unit3 u3;

    public void createUnit1(){      //创建具体的单元 1
        //u1=......
    }
    public void createUnit2(){      //创建具体的单元 2
        //u2=......
    }
    public void createUnit3(){      //创建具体的单元 3
        //u3=......
    }
    public void composite(){        //单元 1、2、3 以某种方式合成具体 Product 对象
        //u1+ u2 + u3
    }

    public static void main(String []args){
        Product p = new Product();
        p.createUnit1();p.createUnit2();p.createUnit3();
```

```
            p.composite();
        }
    }
```

通过测试 main()方法可以知道，当运行完 p.composite()方法后，所需的 Product 对象才最终建立起来。初看一下，本方法解决了复杂类对象的创建问题，层次清晰。但仔细想想，本方法仅解决了一类 Product 对象的创建问题，如果有两类 Product 对象，该如何呢？也许有人会说，采取相同的策略，代码如下。

```
public class Product {
    Unit1 u1;
    Unit1 u2;
    Unit1 u3;
    // 创建第 1 种 Product 对象方法组
    public void createUnit1() {/* u1=..... */
    }
    public void createUnit2() {/* u2=..... */
    }
    public void createUnit3() {/* u3=..... */
    }
    public void composite() {/* u1 + u2 + u3 */
    }

    // 创建第 2 种 Product 对象方法组
    public void createUnitA1() {/* u1=..... */
    }
    public void createUnitA2() {/* u2=..... */
    }
    public void createUnitA3() {/* u3=..... */
    }
    public void compositeA() {/* u1 + u2 + u3 */
    }

    public static void main(String[] args) {
        int type = 1; // 1: 创建第 Product 对象标识
        Product p = new Product();
        if (type == 1) {
            p.createUnit1();
            p.createUnit2();
            p.createUnit3();
            p.composite();
        }
        if (type == 2) {
            p.createUnitA1();
            p.createUnitA2();
            p.createUnitA3();
            p.compositeA();
        }
    }
}
```

可以看出，随着 Product 产品种类的增多或减少，必须修改已有的源代码。这是我们不希望看到的情况，如何解决呢？生成器模式是解决这类问题的重要手段。

4.2 生成器模式

生成器模式也称为建造者模式。生成器模式的意图在于将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。在软件设计中，有时候面临着一个非常复杂的对象的创建工作。这个复杂的对象通常可以分成几个较小的部分，由各个子对象组合出这个复杂对象的过程相对来说比较稳定，但是子对象的创建过程各不相同并且可能面临变化。根据 OOD 中的 OCP 原则，应该对这些子对象的创建过程进行变化封装。

关于创建复杂对象，常规思维与生成器模式思路的关键差别如图 4-1 所示。

（a）常规思路　　（b）生成器模式关键思路

图 4-1　建立复杂对象设计思想对比图

常规思路是在一个类中包含 *n* 种 Product 产品的所有代码，这与 4.1 节中的 Product 类中的代码相吻合。生成器思路是产品类与创建产品的类相分离。产品类仅 1 个，创建产品的类有 *n* 个。由此，可以递进推出生成器模式编程步骤，具体如下。

（1）定义 1 个产品类。

```
public class Unit1{……}
public class Unit2{……}
public class Unit3{……}
public class Product {
    Unit1 u1;
    Unit2 u2;
    Unit3 u3;
}
```

由于不在该类完成 Product 类对象的创建，所以无需显示定义构造方法。

（2）定义 *n* 个生成器 Build 类。

根据语义，生成器是用来生成 Product 对象的，因此一般来说，Product 是生成器类的一个成员变量；根据语义，每创建一个 Product 对象，本质上都需要先创建 Unit1，Unit2，…，UnitN，再把它们组合成所需的 Product 对象，因此需要 *n* 个 createUnit（）方法及一个组合方法 composite()；由于 createUnit()及 composite()是共性，因此可定义共同的生成器类接口，*n* 个生成器类均从此接口派生即可。代码如下。

```
//定义生成器类接口 IBuild
public interface IBuild {
    public void createUnit1();
    public void createUnit2();
    public void createUnit3();
```

```
    public Product composite();                          //返回值是 Product 对象
}

//定义 3 个生成器类
public class BuildProduct implements IBuild {    //生成第一种 Product
    Product p = new Product();                           //Product 是成员变量

    public void createUnit1() {
        //p.u1= ...                                      //创建 Unit1
    }
    public void createUnit2() {
        //p.u2 = ...                                     //创建 Unit2
    }
    public void createUnit3() {
        //p.u3 =           //创建 Unit3
    }
    public Product composite() {
        //...              //关联 Unit,Unit2,Unit3
        return p;          //返回 Product 对象 p
    }
}

public class BuildProduct2 implements IBuild {       //生成第 2 种 Product
    Product p = new Product();//Product 是成员变量

    public void createUnit() {/*p.u = ......  */}    //创建 Unit
    public void createUnit2(){/*p.u2 = ...... */}    //创建 Unit2
    public void createUnit3(){/*p.u3 = ...... */}    //创建 Unit3
    public Product composite() {
        //......               //关联 Unit1, Unit2, Unit3
        return p;              //返回 Product 对象 p
    }
}

public class BuildProduct3 implements IBuild {       //生成第 3 种 Product
    Product p = new Product();//Product 是成员变量

    public void createUnit1() {/*p.u1 = ......  */} //创建 Unit1
    public void createUnit2(){/*p.u2 = ...... */}    //创建 Unit2
    public void createUnit3(){/*p.u3 = ...... */}    //创建 Unit3
    public Product composite() {
        //......           //关联 Unit1, Unit2, Unit3
        return p;          //返回 Product 对象 p
    }
}
```

通过上述代码可知，若需求分析发生变化，只需增加或删除相应的生成器类即可，无需修改已有的类代码。

（3）定义 1 个统一调度类，也叫指挥者（Director）类，是对生成器接口 IBuild 的封装。

该类及简单的测试代码如下。

```
public class Director {
    private IBuild build;
    public Director(IBuild build){
        this.build = build;
    }
    public Product build(){
        build.createUnit1();
        build.createUnit2();
        build.createUnit3();
        return build.composite();
    }

    public static void main(String []args){
        IBuild build = new BuildProduct();
        Director direct = new Director(build);
        Product p = direct.build();
    }
}
```

通过分析上述代码可知，生成器设计模式涉及 4 个关键角色：产品（Product）、抽象生成器（IBuild）、具体生成器（Builder）、指挥者（Director）。四者的 UML 关系图如图 4-2 所示。

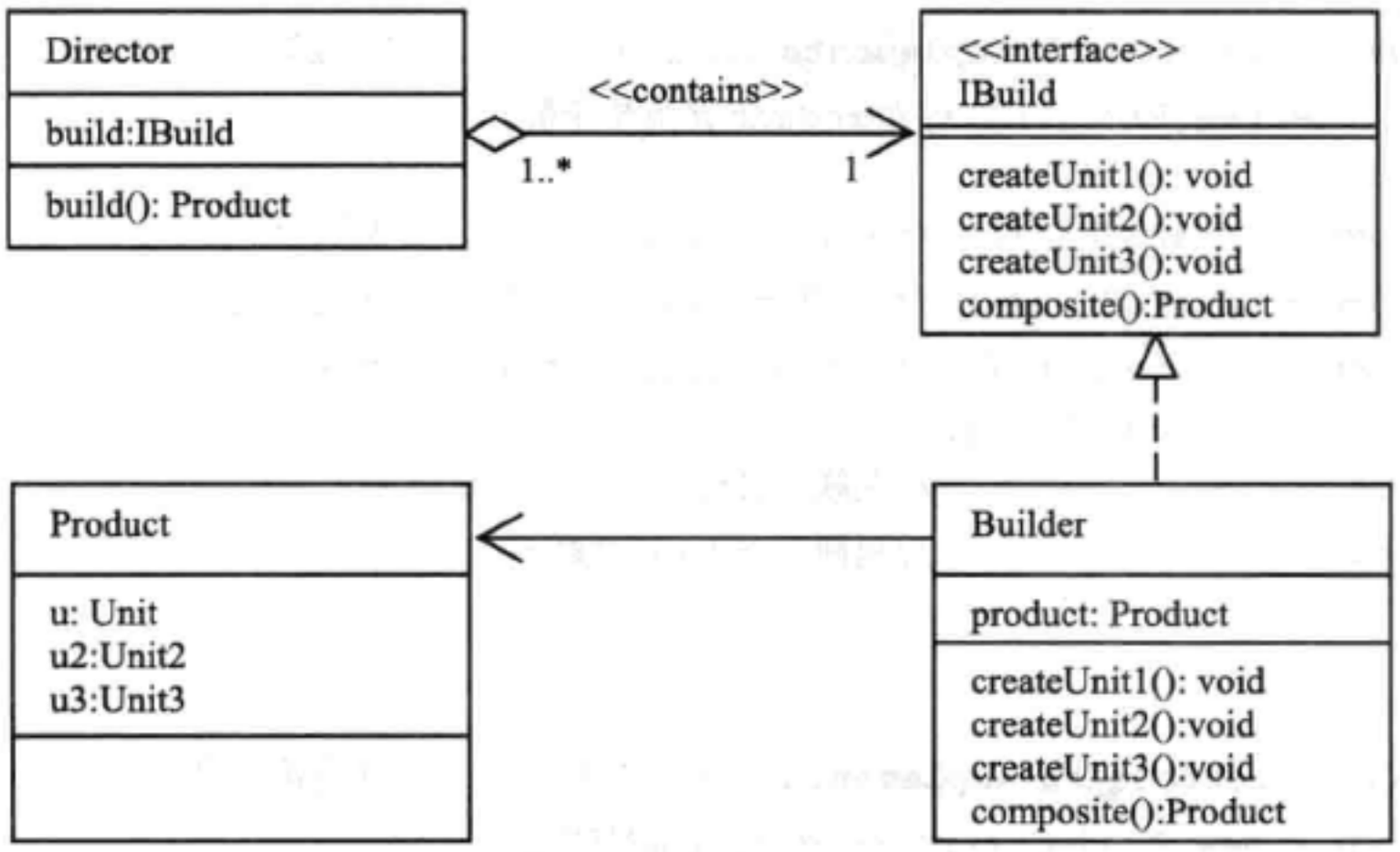


图 4-2　生成器模式类图 1

4.3　深入理解生成器模式

1. 深入理解调度类（指挥者）Director

若想理解好 Director 类，必须理解好 IBuild。与常规接口相比，生成器接口 IBuild 是特殊的，它是一个流程控制接口。该接口中定义的方法必须依照某种顺序运行，一个都不能少。因此，在程序中一定要体现出“流程”这一特点。Director 类即是对“流程”的封装类，其中的 build()方法决定了具体的流程控制过程。

2. 深入理解 IBuild 接口的定义

IBuild 接口清晰地反映了创建产品 Product 的流程。如果将 UML 类图改为图 4-3，又如何呢?

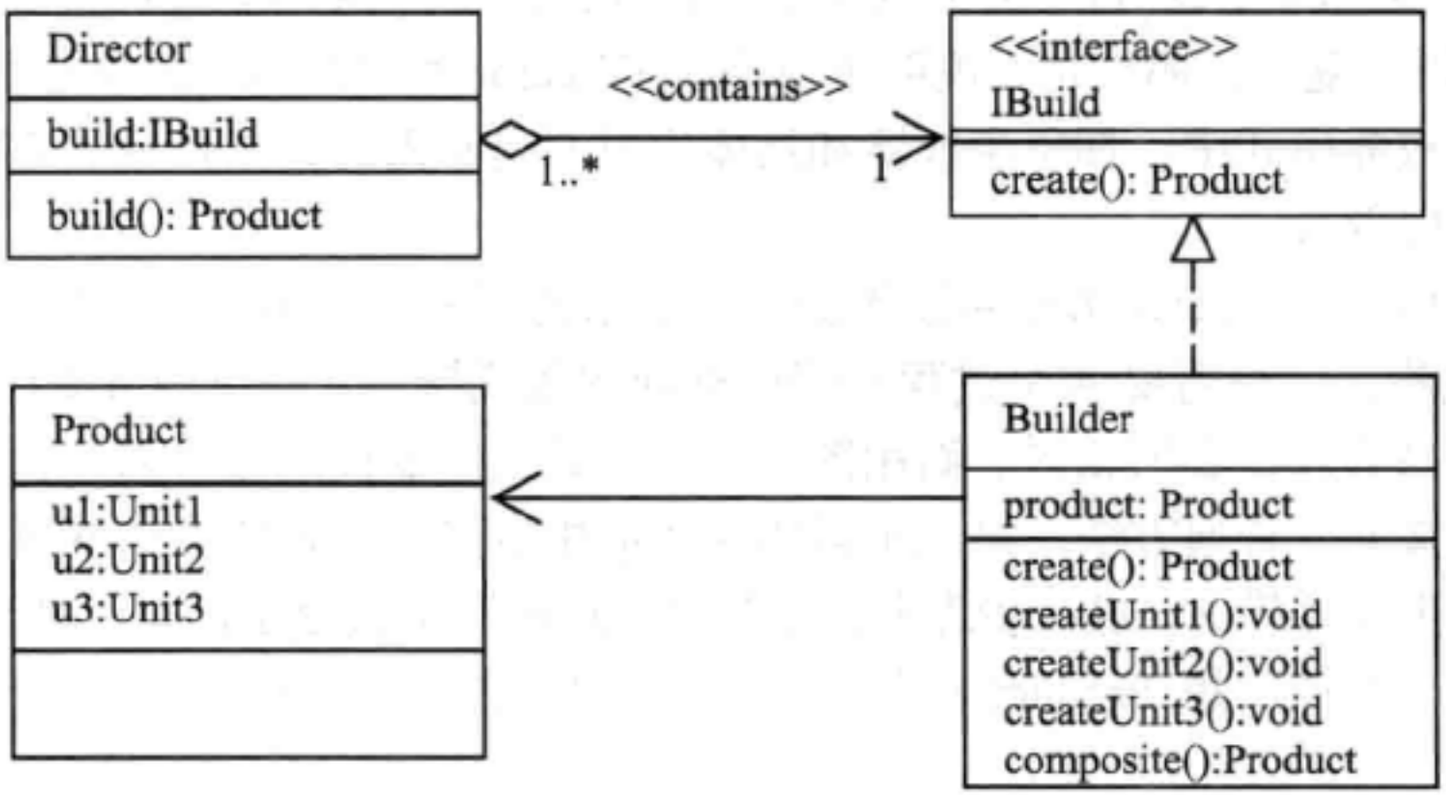


图 4-3　生成器模式类图 2

可以看出，IBuild 接口中仅定义了多态 create()方法，并不能看清创建 Product 需要几个步骤。具体生成器类中重写了多态 create()方法，其中调用了 4 个非多态方法，最终返回了产生的 Product 对象。代码如下。

```
//定义生成器类接口 IBuild
public interface IBuild {
    public Product create();                          //返回值是 Product 对象
}
//定义一个具体的生成器类
public class BuildProduct implements IBuild {    //生成第 2 种 Product
    Product p = new Product();//Product 是成员变量

    public void createUnit1() {/*p.u1 = ......  */}      //创建 Unit1
    public void createUnit2(){/*p.u2 = ...... */}        //创建 Unit2
    public void createUnit3(){/*p.u3 = ...... */}        //创建 Unit3
    public Product composite() {
        //......                    //关联 Unit, Unit2, Unit3
        return p;                   //返回 Product 对象 p
    }
    public Product create(){   //多态方法调用其他创建 Product 对象的方法
        createUnit1(); createUnit2(); createUnit3();
        return composite();
    }
}
//定义指挥者类
public class Director {
    private IBuild build;
    public Director(IBuild build){
        this.build = build;
```

```
        }
        public Product build(){
            return build.create();
        }
    }
```

具体生成器多态的 create()方法中包含了创建 Product 对象的全过程，Director 类中的 build()方法就显得重复了，那么是否说明可以略去 Director 类呢？单纯就本题而言是可以的。也就是说，在生成器模式中，抽象生成器和具体生成器是必需的，指挥者类需要在实际问题中认真考虑，加以取舍。

本方法中利用多态 create()方法可以解决更为复杂的问题。例如要生产两种 Product 产品，一种需要 3 个过程，一种需要 4 个过程；若用标准的生成器模式（见图 4-2）就不行了，因为它要求创建产品的过程必须相同。而用图 4-3 描述的生成器模式是可以的，只需要在第一个具体生成器定义 4 个普通方法，第二个具体生成器中定义 3 个普通方法就可以了。这种模式也可以叫作弱生成器模式。进一步思考，可以把 IBuild 定义成泛型接口，如下面程序所示。所有具体生成器（不仅仅是 Product 产品，也可以是其他需要生成器模式的产品）皆从此接口派生即可。

```
public interface IBuild<T> {
    public T create();   //模板参数 T 是要创建的对象类型
}
```

3. 另一种实现方法

利用 Product 派生类方法，也可以实现类似的生成器功能，类图如图 4-4 所示。

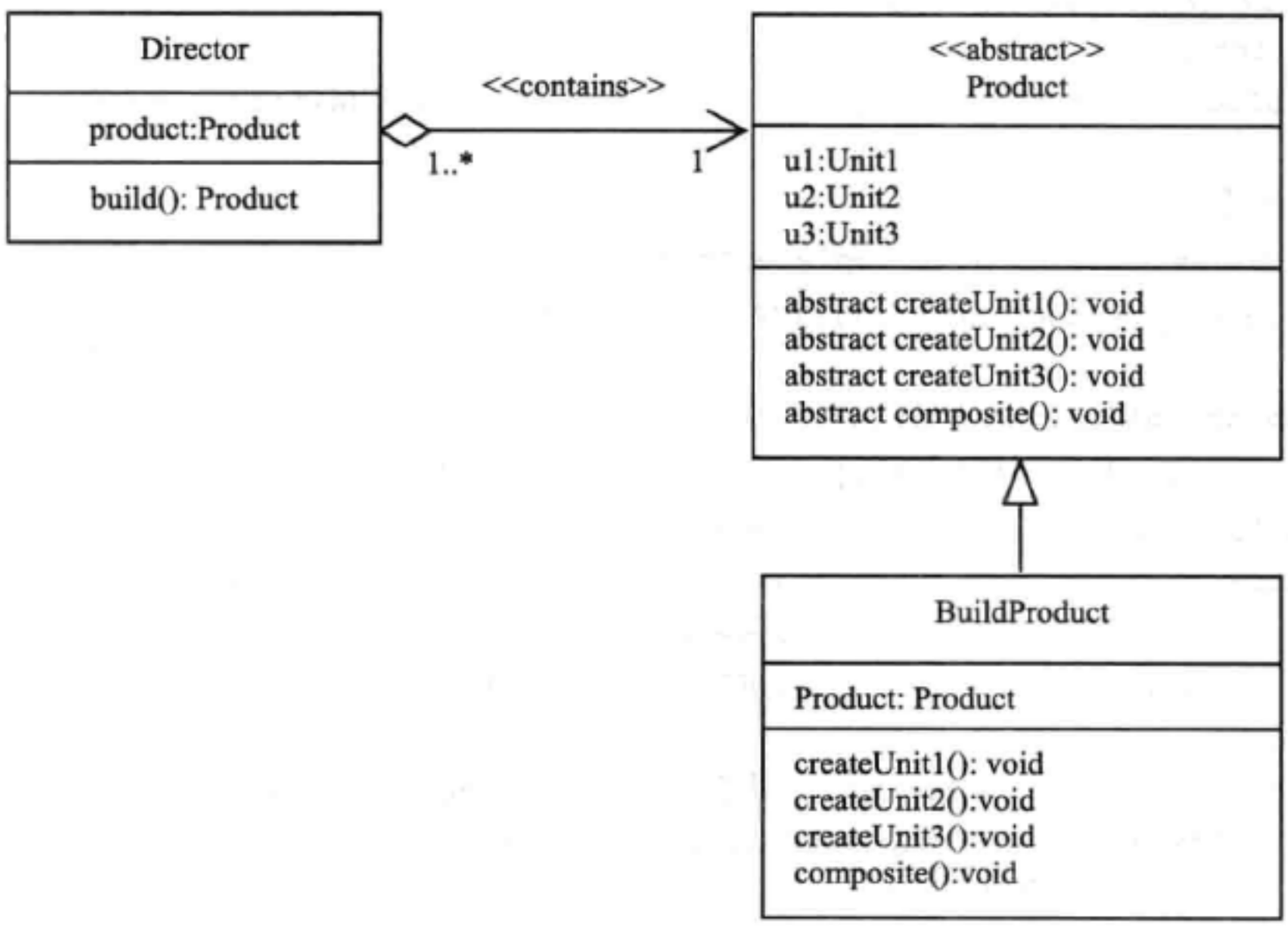


图 4-4　生成器模式类图 3

```
//定义抽象生成器
public abstract class Product {
    Unit1 u1;
    Unit2 u2;
```

```
    Unit3 u3;

    abstract void createUnit1();          //表明子类要创建Unit1
    abstract void createUnit2();          //表明子类要创建Unit2
    abstract void createUnit3();          //表明子类要创建Unit3
    abstract void composite();            //表明子类要组合Unit1、Unit2、Unit3
}
//定义具体生成器
public class BuildProduct extends Product {
    void createUnit1() {/*u1=......*/}
    void createUnit2() {/*u2=......*/}
    void createUnit3() {/*u3=......*/}
    void composite() {/*关联u、u2、u3*/}
}
//定义指挥者类
public class Director {
    Product p;
    public Director(Product p){
        this.p = p;
    }
    void build(){
        p.createUnit1();
        p.createUnit2();
        p.createUnit3();
        p.composite();
    }
}
```

总之，对于生成器模式创建复杂对象而言，主要原则是对象构建过程与表示相分离。这是一个总的思想，实现的具体形式不是一成不变的，或许你也可以写出适合自己应用的生成器模式框架代码来。

4.4 应 用 探 究

1. JavaSE 中生成器的应用

【例 4-1】 通用“更新”功能生成器模式代码。

权限（login 表）是 MIS 系统中的重要功能，不同的角色有不同的功能。如教学管理系统中基本角色有学生(student 表)、教师（teacher 表）等。基本表说明如表 4-1 所示。

表 4-1　　基本表内容说明

login 表				
序号	字段名	说明	关键字	外键
1	User	用户名	√	
2	Pwd	密码		
3	Type	类型:1,学生;2,教师		

续表

student 表

序号	字段名	说明	关键字	外键
1	User	用户名	√	√
2	Name	姓名		
3	Age	年龄		
4	Major	专业		
5	Depart	学院		

teacher 表

序号	字段名	说明	关键字	外键
1	User	用户名	√	√
2	Name	姓名		
3	Age	年龄		
4	Course	主讲课程		
5	Depart	学院		

所有角色的登录用户名及密码信息均在 login 表中，type 为 1 表示是学生，为 2 表示是教师。学生的具体信息在 student 表中，教师的具体信息在 teacher 表中，均通过外键 user 与 login 表关联。一个常用的功能是：管理员为学生和教师在 login 表中分配了用户名及账户，同时在相应的 student 或 teacher 表中建立了关键字为 user 的记录，但其他具体信息如姓名、年龄等均是空的。因此需要学生或教师在登录后首先完善个人信息。本文主要利用生成器模式设计“个人信息完善”的基础代码。

从流程角度来说，更新学生表或教师表是相似的，只是界面显示信息稍有不同。生成器模式是解决同流程、异界面问题的重要手段，代码如下。

（1）界面抽象生成器类 UIBuilder。

```
import javax.swing.*;
public abstract class UIBuilder {
    protected JPanel panel = new JPanel();

    abstract public void addUI();                          //形成界面
    abstract public void registerMsg();                    //注册消息
    abstract public void initialData(String user);         //初始化界面数据
    public JPanel getPanel(){                              //返回界面面板对象
        return panel;
    }
}
```

可以看出，定义的成员变量 panel 是最终要生成的界面对象。对于学生或教师实体而言，形成 panel 界面的过程不同，按钮消息响应不同，界面初始化数据来源不同。因此定义了与之相匹配的三个抽象方法。

（2）具体学生界面生成器类 StudentBuilder。

```
public class StudentBuilder extends UIBuilder implements ActionListener {
```

```
String user;
JTextField   studName = new JTextField(10); //姓名
JTextField   studAge = new JTextField(10);  //年龄
JTextField  studMajor = new JTextField(10);//专业
JTextField  studDepart = new JTextField(10);//学院
JButton  updateBtn = new JButton("更新");//该按钮需注册事件

public void addUI() { //界面生成方法
    JPanel center = new JPanel();
    JPanel south  = new JPanel();
    Box b = Box.createVerticalBox();          //第1列垂直Box对象b
    b.add(new JLabel("姓名"));b.add(Box.createVerticalStrut(8));
    b.add(new JLabel("年龄"));b.add(Box.createVerticalStrut(8));
    b.add(new JLabel("专业"));b.add(Box.createVerticalStrut(8));
    b.add(new JLabel("学院"));b.add(Box.createVerticalStrut(8));
    Box b2 = Box.createVerticalBox();             //第2列垂直Box对象b2
    b2.add(studName);b2.add(Box.createVerticalStrut(8));
    b2.add(studAge); b2.add(Box.createVerticalStrut(8));
    b2.add(studMajor); b2.add(Box.createVerticalStrut(8));
    b2.add(studDepart);b2.add(Box.createVerticalStrut(8));
    center.add(b);center.add(b2);                 //center面板 = b+b2
    south.add(updateBtn);                         //south面板 = updateBtn
    panel.setLayout(new BorderLayout());          //设置panel面板为方位布局管理器
    panel.add(center, BorderLayout.CENTER);       //panel中心方位 = center面板
    panel.add(south, BorderLayout.SOUTH);         //panel南方方位 = south面板
}
public void registerMsg() {
   //消息响应加在本类中，故实现ActionListener接口
   updateBtn.addActionListener(this);
}
public void initialData(String user1) {          //界面数据显示初始化
    this.user=user1
    String strSQL = "select name,age,major,depart from student where user='"
                    +user+ "'";
    DbProc dbobj = new DbProc();                  //数据库操作类
    try{
        dbobj.connect();
        List l = (List)dbobj.executeQuery(strSQL);
        Vector v = (Vector)l.get(0);
        studName.setText((String)v.get(0));       //设置姓名显示编辑框
        studAge.setText((String)v.get(1));
        studMajor.setText((String)v.get(2));
        studDepart.setText((String)v.get(3));
        dbobj.close();
      }
      catch(Exception e){}
  }
public void actionPerformed(ActionEvent arg0) {      // 获得界面数据+更新数据库

      String name = studName.getText();
```

```
String age = studAge.getText();
String major = studMajor.getText();
String depart = studDepart.getText();
String strSQL = "update student set name='" + name + "',age=" + age
        + ",major='" + major + "',depart='" + depart + "'"
        + " where user='" + user + "'";

try {
    DbProc dbobj = new DbProc();
    dbobj.connect();
    dbobj.executeUpdate(strSQL);
    dbobj.close();
} catch (Exception e) {
}
}
```

DbProc 是数据库操作自定义封装类，包括增、删、改、查功能。为了方便，数据库驱动程序字符串、数据源连接字符串直接写在了类中。其实，可以把数据库相关信息存放在配置文件中，通过解析配置文件获得更灵活的数据库操作功能。

本书中的示例全部使用 MySQL 数据库。数据库操作需要将 MySQL 驱动程序（如 connection-java-5.1.17-bin.jar）添加到项目的 classpath 环境变量中；如果是 Java Web 应用，应将驱动程序加入 Web 项目的 WEB-INF/lib 文件夹中。代码如下。

```
//数据库操作封装类 DbProc
public class DbProc {
    private String strDriver = "com.mysql.jdbc.Driver";    //这些信息也可存于配置文件中
private String strDb = "jdbc:mysql://localhost:3306/test";
private String strUser = "root";
private String strPwd = "123456";
private Connection conn;
public Connection connect()throws Exception{
    Class.forName(strDriver);                                    //加载驱动程序
    conn = DriverManager.getConnection(strDb,strUser,strPwd);    //连接数据源
    return conn;
}
public int executeUpdate(String strSQL)throws Exception{         //增、删、改功能
    Statement stm = conn.createStatement();
    int n =  stm.executeUpdate(strSQL);
    stm.close();
    return n;
}

public List executeQuery(String strSQL)throws Exception{         //查询功能
    List l = new Vector();
    Statement stm = conn.createStatement();
    ResultSet rst = stm.executeQuery(strSQL);
    ResultSetMetaData rsmd = rst.getMetaData();
    while(rst.next()){
        Vector unit = new Vector();
        for(int i=1; i<=rsmd.getColumnCount(); i++){
            unit.add(rst.getString(i));
```

```
            }
            l.add(unit);
        }
        return l;
    }
    public void close()throws Exception{                    //关闭数据源连接
        conn.close();
    }
}
```

（3）具体教师界面生成器类 TeacherBuilder。

与 StudentBuilder 类似，可得出 TeacherBuilder 类代码，故略去。

（4）流程指挥者类 Director。

```
import javax.swing.*;
public class Director {
    private UIBuilder build;
    public Director(UIBuilder builder) {
      this.build = builder;
    }
    public JPanel build(String user) {
      build.addUI();                    //初始化界面
      build.registerMsg();              //登记消息
      build.initialData(user);          //填充账号为 user 的初始界面显示数据
      return build.getPanel();
    }
}
```

（5）一个简单的测试类。

上文中（1）~（4）是功能类，本测试类并不是实际应用中的代码，仅是一个简单的仿真，前提条件是在 student 表中有 user 值为“1001”的记录，代码如下。

```
import javax.swing.*;
public class MyTest {
    public static void main(String[] args) {
        JFrame frm = new JFrame();
        UIBuilder ub = new StudentBuilder();    //创建学生生成器
        Director direct = new Director(ub);     //为学生生成器创建指挥者
        JPanel panel = direct.build("1001");    //指挥者创建张同学的更新界面
        frm.add(panel);
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frm.pack();
        frm.setVisible(true);
    }
}
```

以上都是针对 Java 工程应用程序论述的。而随着信息技术的飞速发展，Java Web 程序已是大势所趋，那么如何在 Web 中实现例 4-1 所述的生成器模式基础代码呢？我们知道，Web 程序涉及多种技术。下面主要讨论基于 Ajax 技术的生成器模式代码的实现。

2. 在 Web 项目中的生成器应用

【例 4-2】 在 Web 项目中完成下面的生成器应用，界面如图 4-5 所示。

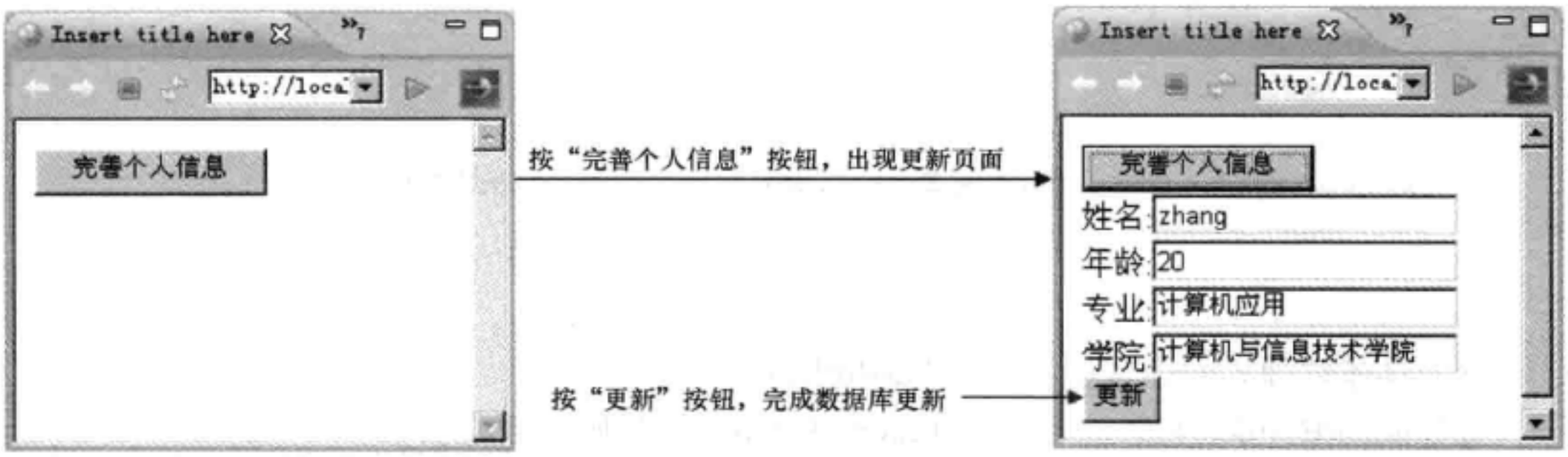


图 4-5　示例初始界面与更新界面

（1）服务器端代码实现。

• 界面抽象生成器接口 IWebBuild。

Java 工程中定义的是抽象类 UIBuilder，主要包含界面生成、消息注册、返回构造好的最终面板对象等功能。但在 Web 程序中，服务器端只能生成待显示的 HTML 字符串，该字符串只有返回客户端 IE 浏览器才能显示出真实的页面；既然在服务器端不能显示界面，就无需在服务器端完成消息的注册。因此，抽象生成器定义成如下接口即可。

```
public interface IWebBuild {
    public void getData(String user);   //从数据库获得账号为 user 的具体数据
    public String getUI();              //形成完全的 HTML 字符串
}
```

• 学生具体生成器类 StudentWebBuild。

```
import java.util.*;
public class StudentWebBuild implements IWebBuild {
    private String name,age,major,depart;
    public void getData(String user) {
        String strSQL = "select name,age,major,depart from student where user='"
                        +user+ "'";
        DbProc dbobj = new DbProc();
        try{
            dbobj.connect();
            List l = (List)dbobj.executeQuery(strSQL);
            dbobj.close();
            Vector v = (Vector)l.get(0);
            name = (String)v.get(0);  age = (String)v.get(1);
            major= (String)v.get(2);  depart = (String)v.get(3);
        }
        catch(Exception e){}
    }
    public String getUI() {
        String s = "姓名:<input type='text' id='name' value='" +name+ "'/><br>" +
                   "年龄:<input type='text' id='age' value='" +age+ "'/><br>" +
                   "专业:<input type='text' id='major' value='" +major+ "'/><br>" +
                   "学院:<input type='text' id='depart' value='" +depart+ "'/><br>" +
                   "<input type='button' id='myupdate' value='更新'>";
```

```
            return s;
        }
}
```

- 教师具体生成器类 TeacherWebBuild。

与学生具体生成器类代码类似，仿写即可，故略去。

- 流程指挥者类 Director。

```
public class Director {
        private IWebBuild build;
        public Director(IWebBuild build) {
          this.build = build;
        }
        public String build(String accountNO) {
          build.getData(accountNO);
          return build.getUI();
        }
}
```

- servlet 请求类 UpdateServlet 类。

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class UpdateServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public UpdateServlet() {
    }
    protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html; charset=utf-8");
            /*为了演示方便，请把下一行注释符号“//”去掉。若在实际应用中，在此处获得登录用户账
            号，替换成相应代码即可。*/
        //request.getSession().setAttribute("user", "1001");

        int type = Integer.parseInt(request.getParameter("mytype"));
        String accountNO = request.getParameter("user");
        IWebBuild obj = null;
        switch(type){          //根据 type 值决定是学生还是教师
        case 1:
            obj = new StudentWebBuild(); break;
        case 2:
            obj = new TeacherWebBuild(); break;
            }
            Director direct = new Director(obj);
            String s = direct.build(accountNO);
            //下一行用 hidden 方式，为把账号传到客户端，以备客户端用
            s += "<input type='hidden' id='account' value='" +user+ "'/>";

            PrintWriter out = response.getWriter();
            out.print(s);
        }
    }
```

服务器端执行流程如下：① 客户端通过 Ajax 技术发送 URL 请求，参数包括 mytype，其值为 1 表示要显示学生信息界面，其值为 2 表示要显示教师信息界面；② UpdateServlet 类代码接收 URL 请求，并解析 mytype 参数，确定选择具体的学生生成器类或教师生成器类；③ 执行生成器流程；④ 向客户端输出 HTML 字符串。

（2）客户器端代码实现。

主要是利用 Ajax 技术及 JavaScript 编码，代码（见 studfunc.js）如下所示。

```
var curobj = null;
var xmlHtpRq = new ActiveXObject("Microsoft.XMLHTTP");//建立 Ajax 异步通讯对象
//为 obj 组件注册 type（如 click）消息，响应函数是 fn
function addEvent(obj, type, fn){
    if(obj && obj.attachEvent){    //IE
        obj.attachEvent("on"+type, fn);
    }
}
//选择学生还是教师类
function select(var type){
    switch(type){
        case 1: curObj=new StudObj(); break;
        case 2: curObj=new TeacherObj(); break;
    }
}
function StudObj(){
}
//Ajax 请求学生信息更新页面
StudObj.prototype.update=function(e){
    url = "updateservlet?mytype=1";
    xmlHtpRq.open("post", url, true);
    xmlHtpRq.onreadystatechange = curobj.update_state;
    xmlHtpRq.send(null);
}
//Ajax 响应学生信息显示页面
StudObj.prototype.update_state = function(){
    if(xmlHtpRq.readyState == 4){
        if(xmlHtpRq.status == 200){
            var obj = document.getElementById("content");
            obj.innerHTML = xmlHtpRq.responseText;
            obj = document.getElementById("myupdate");//完成“更新”按钮消息注册
            addEvent(obj,"click",curobj.updateProc)
        }
    }
}
//Ajax 请求完成更新过程
StudObj.prototype.updateProc = function(e){
    var account = document.getElementById("account").value;
    var name = document.getElementById("name").value;
    var age = document.getElementById("age").value;
    var major = document.getElementById("major").value;
    var depart = document.getElementById("depart").value;
    var strsql = "update student set name='"+name+"',age='"+age+"'," +
            "major='"+major+"',depart='"+depart+"' where user='"+user+"'";
    url = "dbservlet";
```

```
        xmlHtpRq.open("post", url, true);
        xmlHtpRq.onreadystatechange = studobj.updateProc_state;
        xmlHtpRq.send(null);
    }
    //Ajax响应更新完成过程
    StudObj.prototype.updateProc_state = function(){
    }
```

本例中仅写出了有关学生的 JavaScript 类 StudObj，仿此可写出教师的 JavaScript 类 TeacherObj。

由于学生类和教师类都共享定义的 Ajax 异步通讯对象 xmlHtpRq 及消息注册方法 addEvent()，所以把它们分别定义为全局变量和全局方法。

全局方法 select()用于根据 type 值确定全局变量 curobj 是 StudObj 学生对象还是 Teacher 教师对象。

客户端主要包括界面显示及更新数据两个流程。界面显示流程由页面显示请求 update() 及异步应答 update_state()方法组成；更新数据流程指在显示页面上输入相应数据，按“更新”按钮后的行为，由更新请求 updateProc()及更新应答 updateProc_state()方法组成。

着重分析一下 update_state()方法，其关键代码如下所示。

```
var obj = document.getElementById("content");------------------①
obj.innerHTML = xmlHtpRq.responseText; -------------------------②
obj = document.getElementById("myupdate");//完成“更新”按钮消息注册--------③
addEvent(obj,"click",curobj.updateProc); ------------------------④
```

第②行显示了真实的更新页面，第④行完成了“更新”按钮的消息注册。到此为止，客户端代码加上服务器端代码才基本完成了 Web 编程下的生成器模式的基本框架。

updateProc()方法的具体功能是采集界面已有数据，形成更新的 SQL 语句，并把此语句传送到服务器端，服务器端执行该 SQL 语句，完成数据的更新。因此，在服务器端定义了 DbServlet 类，代码如下。

```
    import java.io.*;
    import javax.servlet.*;
    import javax.servlet.http.*;
    public class DbServlet extends HttpServlet implements Servlet {
        private static final long serialVersionUID = 1L;
        public DbServlet() {
            super();
        }
        protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
            request.setCharacterEncoding("utf-8");
            response.setContentType("text/html; charset=utf-8");
            String strsql = request.getParameter("strsql");
            DbProc obj = new DbProc();//同前文论述的DbProc类一致
            try{
                obj.connect();
                obj.executeUpdate(strsql);
                obj.close();
            }
            catch(Exception e){}
        }
    }
```

（3）一个简单的测试 JSP。

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-
8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/
TR/html4/loose.dtd">
<html>
<head>
<script type="text/javascript" src="studfunc.js"></script>
<script type="text/javascript">
    function window.onload(){
        select(1); //选择学生生成器
        var obj = document.getElementById("studedit");
        addEvent(obj,"click",curobj.update);
    }
</script>
</head>
<body>
<div><input type="button" id="studedit" value="完善个人信息" /></div>
<div id="content"></div>
</body>
</html>
```

3. 应用生成器实现数据导入

【例 4-3】 数据库数据导入功能类。

例如，将不同格式学生成绩文件信息导入数据库表 score 中，score 表结构如表 4-2 所示。

表 4-2　　学生成绩表 score 结构

序　号	字 段 名	说　明	关 键 字
1	setudno	学号	√
2	name	姓名	
3	score	成绩	

分析：尽管学生成绩文件格式可能不同，但导入 score 表中的流程是相同的，如图 4-6 所示。

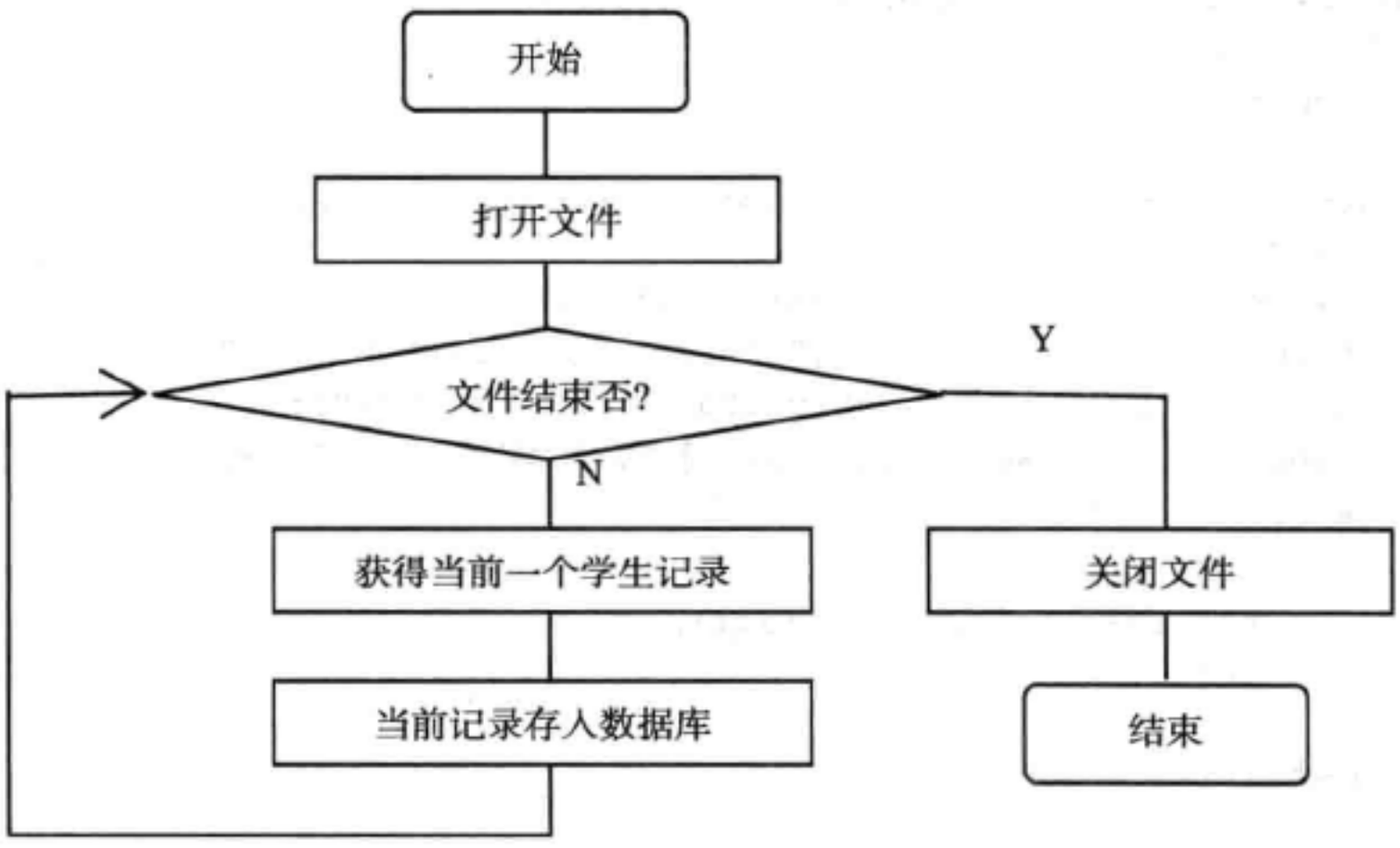


图 4-6　学生信息数据文件导入数据库表流程图

由此，可得采用生成器模式的关键代码如下。

（1）抽象生成器 AbstractBuild 类。

```
public abstract class AbstractBuild {
    public abstract boolean open(String strPath);    //打开文件
    public abstract boolean hasNext();               //文件还有记录吗?
    public abstract Score next();                    //取下一条记录
    public abstract void close();                    //关闭文件
    public boolean saveToDb(Score s){                //将该条记录存入数据库
        String strSQL = "insert into score values('" +s.getStudno()+ "'," +
                        "'" +s.getName()+ "'," +s.getScore()+ ")";
        //数据库操作功能类，与之前的 DbProc 一致
        DbProc dbobj = new DbProc();
        try{
            dbobj.connect();
            dbobj.executeUpdate(strSQL);
            dbobj.close();
        }
        catch(Exception e) {}
        return true;
    }
}
```

对于不同格式文件，open()、hasNext()、next()、close()方法是不同的，所以把它们定义成抽象方法。着重注意 next()方法，它的返回值是统一的 Score 对象，表明在 next()方法内部处理随格式不同而不同，但出口是相同的。因此，可以把数据保存到数据库功能做成通用方法 saveToDb，无需把该函数定义成抽象方法。Score 类代码如下。

```
public class Score {                //与数据库表 score 对应
    private String studno;          //学号
    private String name;            //姓名
    private int score;              //成绩
    public String getStudno() {
        return studno;
    }
    public String getName() {
        return name;
    }
    public int getScore() {
        return score;
    }
    public void setStudno(String studno) {
        this.studno = studno;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setScore(int score) {
        this.score = score;
    }
}
```

（2）指挥者类 Director。

```
public class Director {
```

```
AbstractBuild build;
public Director(AbstractBuild build){
    this.build = build;
}
public void build(String strPath){
    build.open(strPath);                    //打开文件
    while(build.hasNext()){                 //还有记录吗? 若有，则
        Score s = build.next();             //得到一条记录，并最终转化为统一的 Student 对象
        build.saveToDb(s);                  //保存到数据库
    }
        build.close();                      //关闭文件
    }
}
```

按常规论述来说，这部分应是某具体生成器代码的讲解。但笔者把 Director 提前，旨在加深对 Director 的理解，即 build()方法是封装复杂流程算法的。本例中的算法与图 4-6 是一致的，调用的方法有抽象的，如 open()、hasNext()、next()、close()方法；也有非抽象的共享方法，如 saveToDb()方法。可以看出，抽象生成器定义的流程方法层次也是多种多样的，有并列的，如 open()、close()方法；也有循环嵌套的，如 hasNext()、next()方法。进而可以得出，如果流程算法是完善的，就可从中提取出抽象生成器所定义的各个方法。而且，该复杂算法即是指挥者 Director 类中 build()方法要体现出的内容。

（3）一个具体生成器类 BinaryBuild。

假设学生成绩信息文件 score.dat 是按结构体存储的，定义如下：学号，10 字节；姓名，10 字节；成绩，short。也就是说，每个学生占有 10+10+2=22 字节。之所以用二进制文件而没有用简单的文本文件，一方面希望读者在学习设计模式的同时，加强基本功的训练，另一方面是因为 Java 语言没有对结构体专门操作的方法，而结构体操作在实践中是经常遇到的，这也促使每个研究人员去探索，或许读者也能提出一个圆满的结构体操作解决方案来。

假设学号占满 10 个字节，姓名定义成 10 个字节存储，但未必占满 10 个字节，成绩是 short 类型，占 2 个字节。BuildBuild 类代码如下。

```
import java.io.*;
public class BinaryBuild extends AbstractBuild {
    //数组表明:学号 10 个字节，姓名:10 个字节，成绩:2 个字节
    int unitsize[] = {10,10,2};
    RandomAccessFile in;
    long fileLength;//文件总长度
    long current;   //目前读文件的字节数
    byte buffer[];  //学生信息缓冲区

public BinaryBuild(){
    int size=0;
    for(int i=0; i<unitsize.length; i++)
        size += unitsize[i]; //求每个学生占用的字节总数
    buffer = new byte[size]; //每个学生信息占用的缓冲区
}
public boolean open(String strPath){
    try{
        File file = new File(strPath);
        in = new RandomAccessFile(file,"rw");
```

```
            fileLength = file.length();     //求文件总长度
        }
        catch(Exception e){ }
        return true;
    }
    public boolean hasNext(){
        return current < fileLength;        //当前文件位置<文件长度，表明未到文件尾
    }
    Public Score next(){
        Student obj = new Student();
        try{
            in.read(buffer);
            current += buffer.length;
            obj.setStudno(new String(buffer,0,10));          //设置学号
            int n = 10;
            while(buffer[n]!=0) n++;
            obj.setName(new String(buffer,10,n));            //设置姓名
            obj.setScore((int)buffer[21]*256+buffer[20]);    //设置成绩
        }
        catch(Exception e){}
        return obj;
    }
    public void close(){
        try{
            in.close();
        }
        catch(Exception e){    }
    }
```

本程序运行，需要一个二进制的 score.dat 文件，这个文件可以通过 FileOutputStream 类和 DataOutputStream 类来创建，相应地通过 FileInputStream 类和 DataInputStream 类来读取。下面给出的是创建文件的二进制代码。

```
// 创建 score.dat 文件的代码
public static void main(String[] args) {
        Score[] scores = new Score[3];
        scores[0] = new Score("0011000001", "zhang3", 89);
        scores[1] = new Score("0011000011", "li4", 69);
        scores[2] = new Score("0011000020", "feng1", 44);
        byte[] name = new byte[10];
        byte[] mid = null;
        FileOutputStream fos;
        try {
        fos = new FileOutputStream("d:/score.dat");
        DataOutputStream dos = new DataOutputStream(fos);
        for (int i = 0; i < scores.length; i++) {
            for (int j = 0; j < name.length; j++) {
                name[j] = 0;
            }
            mid = scores[i].getName().getBytes();
            for (int k = 0; k < mid.length; k++)
                name[k] = mid[k];
            dos.write(scores[i].getStudno().getBytes());
            dos.write(name);
```

```
            dos.writeInt(scores[i].getScore());
        }
        dos.close();
        fos.close();
        //下面是读文件代码
        FileInputStream fis = new FileInputStream("d:/score.dat");
        DataInputStream dis= new DataInputStream(fis);
        for(int i=0;i<scores.length;i++) {
            byte[] buf1=new byte[10];
            int len=dis.read(buf1);
            System.out.println(new String(buf1,0,len,"GBK"));
            byte[] buf2=new byte[10];
            len=dis.read(buf2);
            System.out.println(new String(buf2,0,len,"GBK"));
            System.out.println(dis.readInt());
        }
        dis.close();
        fis.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

第5章　观察者模式

5.1　问题的提出

在生活实际中，经常会遇到多种对象关注一个对象数据变化的情况。例如，生活中有温度记录仪，当温度发生变化时，需要完成如下功能：记录温度日志，显示温度变化曲线，当温度越界时扬声器发出声音。可能会写出以下程序片段。

```
while(温度变化) {
        记录温度日志;
        显示温度变化曲线;
        当温度越界时扬声器发出声音;
}
```

这种方法把所有功能集成在一起，当需求分析发生变化，例如，增加新的温度监测功能或舍去某一监测功能时，程序都得修改，这是我们所不希望的结果。观察者设计模式是解决这类问题的有效方法。

5.2　观察者模式

观察者设计模式适合解决多种对象跟踪一个对象数据变化的程序结构问题，有一个称作“主题”的对象和若干个称作“观察者”的对象。与第 5.1 节介绍知识对比：有一个主题数据——温度，3 个观察者——温度日志、温度曲线、温度报警。因此观察者设计模式涉及两种角色：主题和观察者。

观察者设计模式可以从以下递推中得出一些重要结论。

- 主题要知道有哪些观察者对其进行监测，因此主题类中一定有一个集合类成员变量，包含了观察者的对象集合。
- 既然包含了观察者的对象集合，那么，观察者一定是多态的，有共同的父类接口。
- 主题完成的主要功能是：可以添加观察者，可以撤销观察者，可以向观察者发消息，引起观察者响应。这三个功能是固定的，因此主题类可以从固定的接口派生。

因此，编制观察者设计模式，要完成以下功能类的编制。

- 主题 ISubject 接口定义。
- 主题类编制。

- 观察者接口 IObserver 定义。
- 观察者类实现。

观察者设计模式典型的 UML 类图如图 5-1 所示。

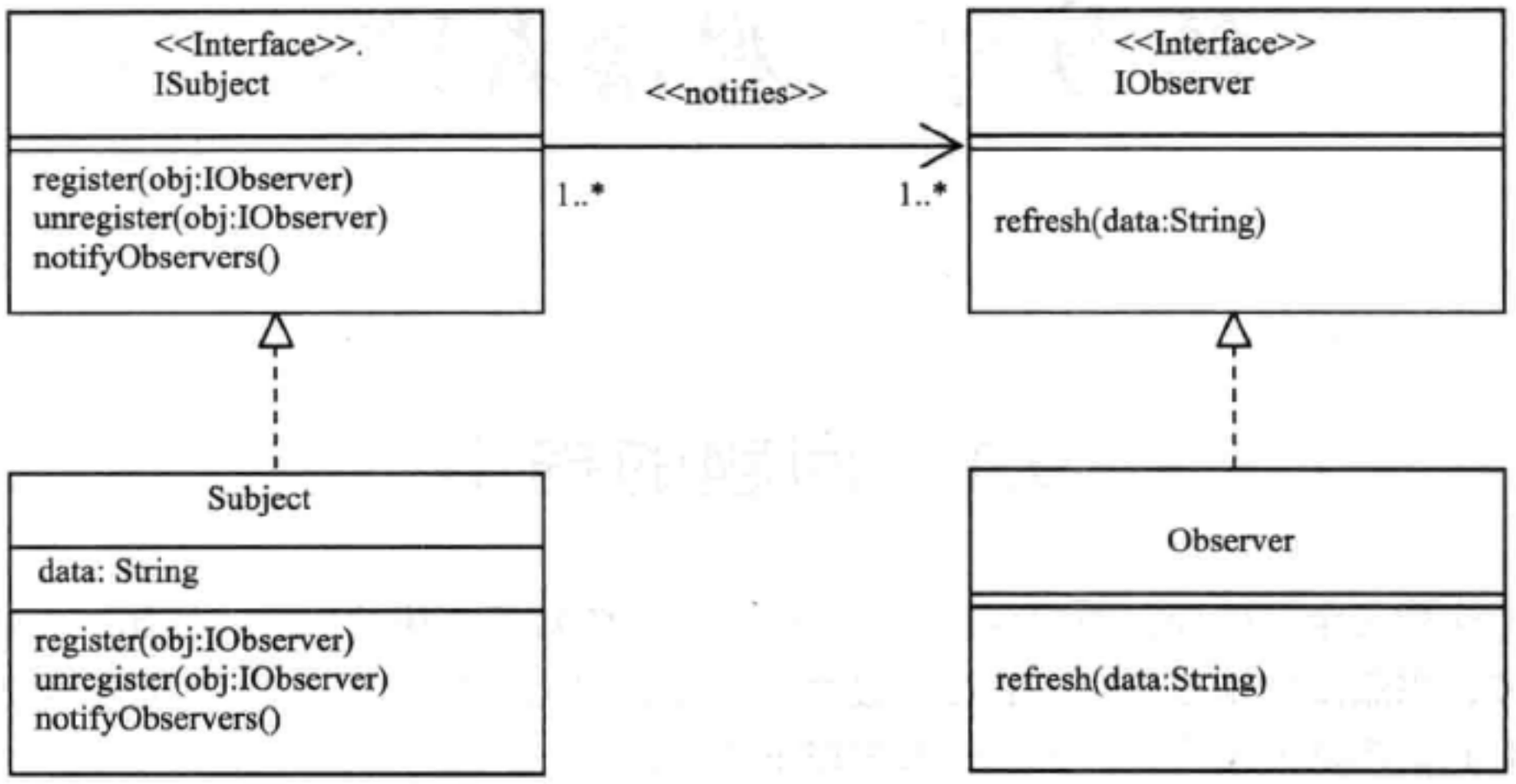


图 5-1　观察者设计模式类图

关键代码如下所示。

（1）观察者接口 IObserver。

```
public interface IObserver {
    public void refresh(String data);
}
```

（2）主题接口 ISubject。

```
public interface ISubject {
    public void register(IObserver obs);          //注册观察者
    public void unregister(IObserver obs);        //撤销观察者
    public void notifyObservers();                //通知所有观察者
}
```

（3）主题实现类 Subject。

```
public class Subject implements ISubject {
    private Vector<IObserver> vec = new Vector();    //观察者维护向量
    private String data;                             //主题中心数据

    public String getData() {
        return data;
    }
    public void setData(String data) {               //主题注册（添加）
        this.data = data;
    }
    public void register(IObserver obs) {            //主题注册（添加）观察者
        vec.add(obs);
    }
```

```
public void unregister(IObserver obs) {          //主题撤销（删除）观察者
    if(vec.contains(obs))
        vec.remove(obs);
}
public void notifyObservers(){                   //主题通知所有观察者进行数据响应
    for(int i=0; i<vec.size(); i++){
        IObserver obs = vec.get(i);
        obs.refresh(data);
    }
}
```

主题实现类 Subject 是观察者设计模式中最重要的一个类，包含了观察者对象的维护向量 vec 以及主题中心数据 data 变量与具体观察者对象的关联方法（通过 nitofyObservers()）。也就是说，从此类出发，可以更深刻地理解 ISubject 为什么定义了 3 个方法、IObserver 接口为什么定义了 1 个方法。

（4）一个具体观察者类 Observer。

```
public class Observer implements IObserver {
    public void refresh(String data) {
        System.out.println("I have received the data:" +data);
    }
}
```

（5）一个简单的测试类 Test。

```
public class Test {
    public static void main(String[] args) {
        IObserver obs = new Observer();      //定义观察者对象
        Subject subject = new Subject();     //定义主题对象
        subject.register(obs);               //主题添加观察者
        subject.setData("hello");            //主题中心数据发生变动
        subject.notifyObservers();           //通知所有观察者进行数据响应
    }
}
```

该段代码的含义是：当主题中心数据变化（通过 setData 方法）后，主题类 subject 要调用 notifyObservers()方法，通知所有观察者对象接收数据并进行数据响应。

5.3　深入理解观察者模式

1. 深入理解 ISubject、IObserver 接口

上文中的 Subject 类中的中心数据 data 是 String 类型的，IObserver 接口中定义的 refresh()方法参数类型因此也是 String 类型的。若 data 改为其他类型，则 IObserver 接口等相关代码都需要修改。其实，只要把 ISubject、IObserver 接口改为泛型接口就可以了，代码如下。

```
//观察者泛型接口 IObserver
public interface IObserver<T> {
    public void refresh(T data);
```

```
}
//主题泛型接口 ISubject
public interface ISubject<T>{
    public void register(IObserver<T> obs);
    public void unregister(IObserver<T> obs);
    public void notifyObservers();
}
```

当把 ISubject、IObserver 接口修改为泛型接口后，要求参数 T 必须是类类型，不能是基本数据类型。如不能是 int，但可以是 Integer 类型。

2. “推”数据与“拉”数据

推数据方式是指具体主题将变化后的数据全部交给具体观察者，即将变化后的数据、直接传递给具体观察者用于更新数据。从 5.2 节中接口 IObserver 定义就可看出这一点，代码如下。

```
public interface IObserver {
    public void refresh(String data);
}
```

可以看出，主题对象直接将数据传送给观察者对象，这是“推”数据方式的最大特点。与之对比，“拉”数据方式的特点是观察者对象可间接获得变化后的主题数据，观察者自己把数据“拉”过来。5.2 节示例修改为“拉”数据方式的代码如下。

```
public interface IObserver {
    public void refresh(ISubject subject);
}
public interface ISubject {                    //同 5.2 节
    public void register(IObserver obs);       //注册观察者
    public void unregister(IObserver obs);     //撤销观察者
    public void notifyObservers();             //通知所有观察者

}
public class Subject implements ISubject {
    //其他所有代码同 5.2 节
    public void notifyObservers(){
        for(int i=0; i<vec.size(); i++){
            IObserver obs = vec.get(i);
            obs.refresh(this);                 //代替原来的 refresh(data)
        }
    }
}
public class Observer implements IObserver {
    public void refresh(ISubject obj) {
        Subject subject = (Subject)obj;        //必须进行强制转换
        System.out.println("I have received the data:" +obj.getData());
    }
}
```

主要将观察者接口 IObserver 中的方法修改为 refresh(ISubject subject)。可推测出：具体观察者子类对象一定能获得主体 Subject 对象，当然也就能间接访问主体对象的变量了。从此观点出发，就容易理解上述 notifyObservers()与 refresh()方法代码的修改情况。

“拉”数据方式观察者设计模式 UML 框图如图 5-2 所示。

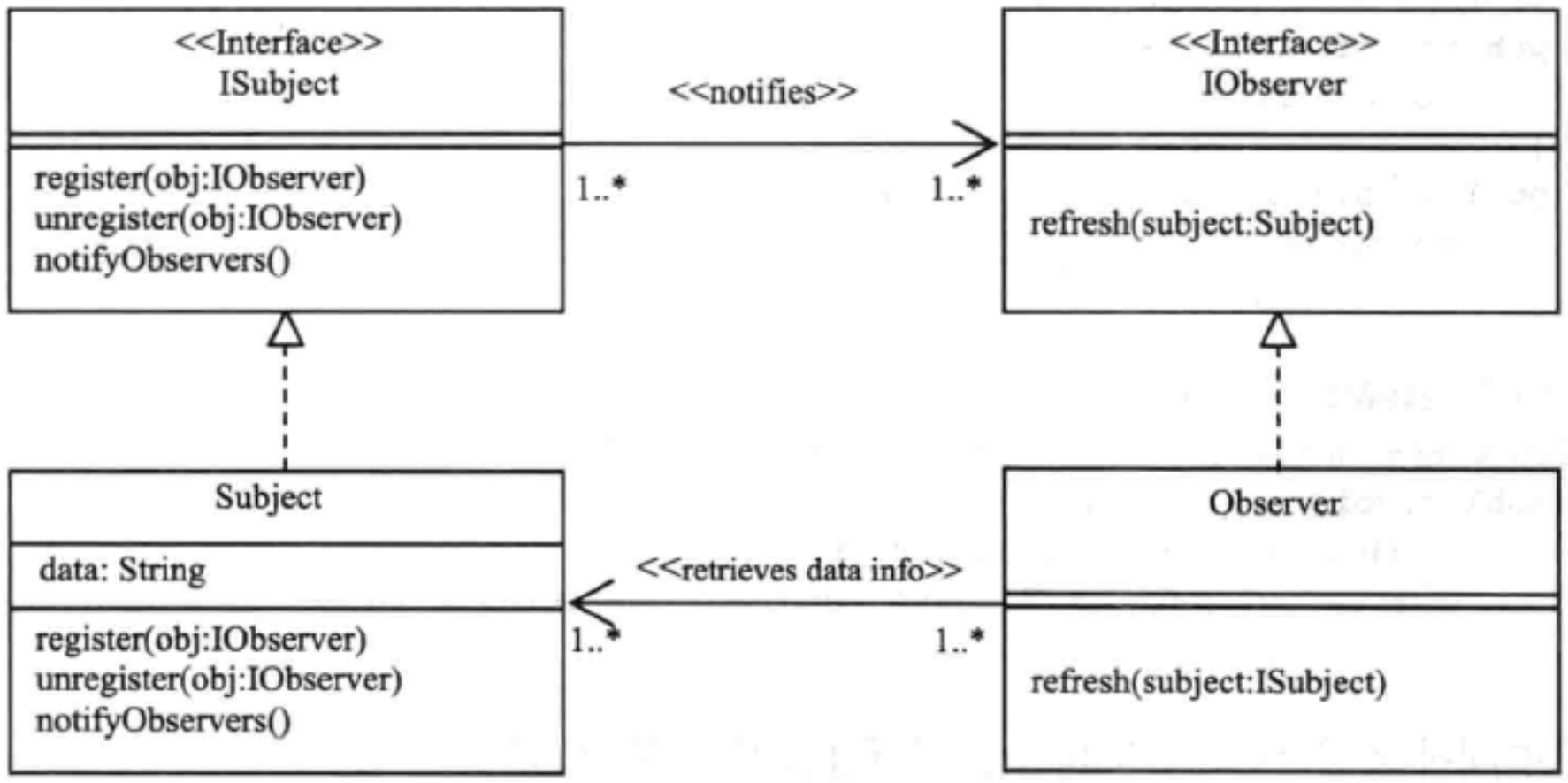


图 5-2　“拉”数据方式观察者设计模式 UML 框图

3. 增加抽象类层 AbstractSubject

假设有多个主题类，按 5.2 节来说，每个主题类都要重写 register()、unregister()、notifyObservers()方法。又假设这三个方法的代码恰巧是相同的（这种可能性是很大的，因为它们都是通用方法），那么每个主题类的代码就显得重复了，用中间层类来解决代码重复问题是一个较好的方法。代码如下。

```
public interface ISubject {                          //同 5.2 节
    public void register(IObserver obs);    //注册观察者
    public void unregister(IObserver obs); //撤销观察者
    public void notifyObservers();          //通知所有观察者
}
public interface IObserver{
    public void refresh(ISubject obj);      //表明将采用"拉"数据方式
}
//增加的抽象层类 AbstractSubject
public class AbstractSubject implements ISubject {
    Vector<IObserver> vec = new Vector();
    public void register(IObserver obs) {
        vec.add(obs);
    }
    public void unregister(IObserver obs) {
        if(vec.contains(obs))
            vec.remove(obs);
    }
    public void notifyObservers(){
        for(int i=0; i<vec.size(); i++){
            IObserver obs = vec.get(i);
            obs.refresh(this);
        }
    }
}
//派生主体类 Subject
```

```
public class Subject extends AbstractSubject {
    private String data;
    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;
    }
}
//一个具体观察者类 Observer
public class Observer implements IObserver {
    public void refresh(ISubject obj) {
        Subject subject = (Subject)obj;
        System.out.println("I have received the data:" +subject.getData());
    }
}
```

有了中间抽象层 AbstractSubject，灵活了具体主题类的编制。使用

```
class XXXSubject extends AbstractSubject{……}
```

表明 register()、unregister()、notifyObservers()方法与 AbstractSubject 类中的对应方法是一致的；还可以使用

```
class XXXSubject implements ISubject{……}
```

表明 3 个方法必须重写。

虽然 AbstractSubject 类中无抽象方法，但定义成了抽象类。这是因为从语义角度上来说，该类不是一个完整的主题类，它缺少主题数据，因此把它定义成抽象类。

4. 避免重复添加同一类型的观察者对象

如果 5.2 节测试类代码改为如下：

```
public class Test {
    public static void main(String[] args) {
        IObserver obs = new Observer();
        IObserver obs2 = new Observer();
        Subject subject = new Subject();
        subject.register(obs);
        subject.register(obs2);

        subject.setData("hello");
        subject.notifyObservers();
    }
}
```

可以看出，obs、obs2 观察者对象类型是相同的，都是 Observer，这两个相同类型的观察者对象都正确地加到了主题对象中。但有些情况下这是不允许的，要求禁止主题对象添加相同的观察者对象。因此，在主题对象添加观察者对象前，应先进行查询，然后判断是否添加观察者对象，register()方法修改后的代码如下。

```
public class Subject implements ISubject {
    public void register(IObserver obs) {          //主题注册（添加）观察者
        if(!vec.contains(obs))                     //如果向量中不包含 obs 观察者
            vec.add(obs);                          //则添加
    }
```

```
        //其他代码略
    }
```

但是 Vector 类中的 contains()方法默认是物理查询。由于 Test 类中的 obs、obs2 虽然都是 Observer 对象，但它们的物理地址是不同的，因此仍添加到了 vec 向量中。这与我们的初衷是不一致的，如何解决呢？基本思想是必须实现自定义查询功能。打开 JDK 源程序压缩包，会发现，contains()方法调用了 indexOf()方法，因此加深理解 indexOf()是非常重要的。其源码如下。

```
public synchronized int indexOf(Object o, int index) {
    if (o == null) {
        for (int i = index ; i < elementCount ; i++)
         if (elementData[i]==null)
            return i;
    } else {
        for (int i = index ; i < elementCount ; i++)
         if (o.equals(elementData[i])) //着重注意这一行语句
            return i;
    }
    return -1;
}
```

着重注意斜体加深的代码 *o.equals(elementData[i])*，两个元素相等是由 equals()决定的，因此，只要重载传入参数 o 中的 equals()就可以了。由于 elementData[i]也是观察者类的对象，所以 equals()方法中的参数原型也明确了。以两个具体观察者为例，功能类代码如下。

```
public interface ISubject {  //同 5.2 节
    public void register(IObserver obs);          // 注册观察者
    public void unregister(IObserver obs);        // 撤销观察者
    public void notifyObservers();                // 通知所有观察者
}
public interface IObserver {
    public int getMark();
    public void refresh(String data);
}
public class Subject implements ISubject {
    public void register(IObserver obs) {
        if(!vec.contains(obs))
            vec.add(obs);
    }
    //其他代码同 5.2
}
//第 1 个观察者类 Observer
public class Observer implements IObserver {
    private final int MARK = 1;
    public int getMark() {
        return MARK;
    }
    public boolean equals(Object arg0) {
        Observer obj = (Observer)arg0;
        return obj.getMark() == MARK;
    }
```

```
    public void refresh(String data) {
        System.out.println("I have received the data data:" +data);
    }
}
//第 2 个观察者类 Observer2
public class Observer2 implements IObserver{
    private final int MARK = 2;
    public int getMark() {
        return MARK;
    }
    public boolean equals(Object arg0) {
        Observer obj = (Observer)arg0;
        return obj.getMark() == MARK;
    }
    public void refresh(String data) {
        System.out.println("I have received the data data:" +data);
    }
}
```

程序的测试类与前一个例子基本相同，读者可以自行完成。

关键思路是：在每个观察者类中增加一个标识常量 MARK，不同类型的观察者对象中 MARK 常量值是不同的。本例中，Observer 中 MARK 值为 1，Observer2 中 MARK 值为 2 。由于主题类 Subject 中 register()方法参数 obs 是 IObserver 类型，是多态表示，因此在 IObserver 接口中必须增加多态方法 getMark()，用于获得观察者对象中的 MARK 值。

5. 反射技术的应用

将观察者类信息封装在 XML 配置文件中，从而利用反射技术可以动态加载观察者对象。配置文件采用键—值配对形式，值对应的是具体观察者的类名称。由于键是关键字，不能重复，为了编程方便，键采用“统一前缀+流水号”的形式，配置文件示例如表 5-1 所示。

表 5-1　　观察者设计模式配置文件示例

XML 配置文件	说　明
<?xml version="1.0" encoding="UTF-8" standalone="no"?> <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"> <properties> <comment>Observer</comment> <entry key="observer1">Observer</entry> <entry key="observer2">Observer2</entry> </properties>	键前缀“observer”， 键流水号“1，2，……”

具体程序代码如下。

```
public interface IObserver {//同 5.2 节}
public interface ISubject {
    public void register(String strXMLPath);          //表明从配置文件加载观察者对象
    public void unregister(IObserver obs);
    public void notifyObservers();
}
//主体类 Subject
public class Subject implements ISubject {
    //其他代码同 5.2 节
```

```
public void register(String strXMLPath){
    String prefix = "observer" ;
    String observeClassName = null;
    Properties p = new Properties();
    try{
        FileInputStream in = new FileInputStream(strXMLPath);
        p.loadFromXML(in);
        int n = 1;
        while((observeClassName=p.getProperty(prefix+n)) != null){
            Constructor c = Class.forName(observeClassName).getConstructor();
            IObserver obs =
             (IObserver)Class.forName(observeClassName).newInstance();
            vec.add(obs);
            n ++;
        }
        in.close();
    }
    catch(Exception e){
        e.printStackTrace();
    }
  }
}
//一个具体观察者类 Observer
public class Observer implements IObserver{//同 5.2 节
}
public class Observer2 implements IObserver{//同 5.2 节
}
//一个简单测试类 Test
public class Test {
    public static void main(String[] args)throws Exception {
        Subject subject = new Subject();                    //定义主题对象
        subject.register("d:/config/info5.xml");    //主题通过配置文件加载观察者对象
        subject.setData("hello");                           //主题数据变化了
        subject.notifyObservers();                          //通知各观察者对象进行数据响应
    }
}
```

如果保证程序顺利运行，还要实现另一个观察者，这样才和前面的 XML 文件描述相匹配。

5.4　JDK 中的观察者设计模式

由于观察者设计模式中主题类功能及观察者接口定义内容的稳定性，JDK 的 java.util 包提供了系统的主题类 Observable 及观察者接口 Observer，其 UML 类图如图 5-3 所示。

很明显，Observer 相当于 5.2 节中的 IObserver 观察者接口类，其中的 update()方法中第一个参数是 Observable 类型，表明采用的是“拉”数据方式；Observable 相当于 5.2 节中的主题类 Subject，addObserver()、deleteObserver()、notifyObservers()三个方法分别表示“添加、

删除、通知”观察者对象功能。XXXchanged()方法主要是设置或获得 changed 成员变量的值或状态，changed 为 true 时表明主题中心数据发生了变化。

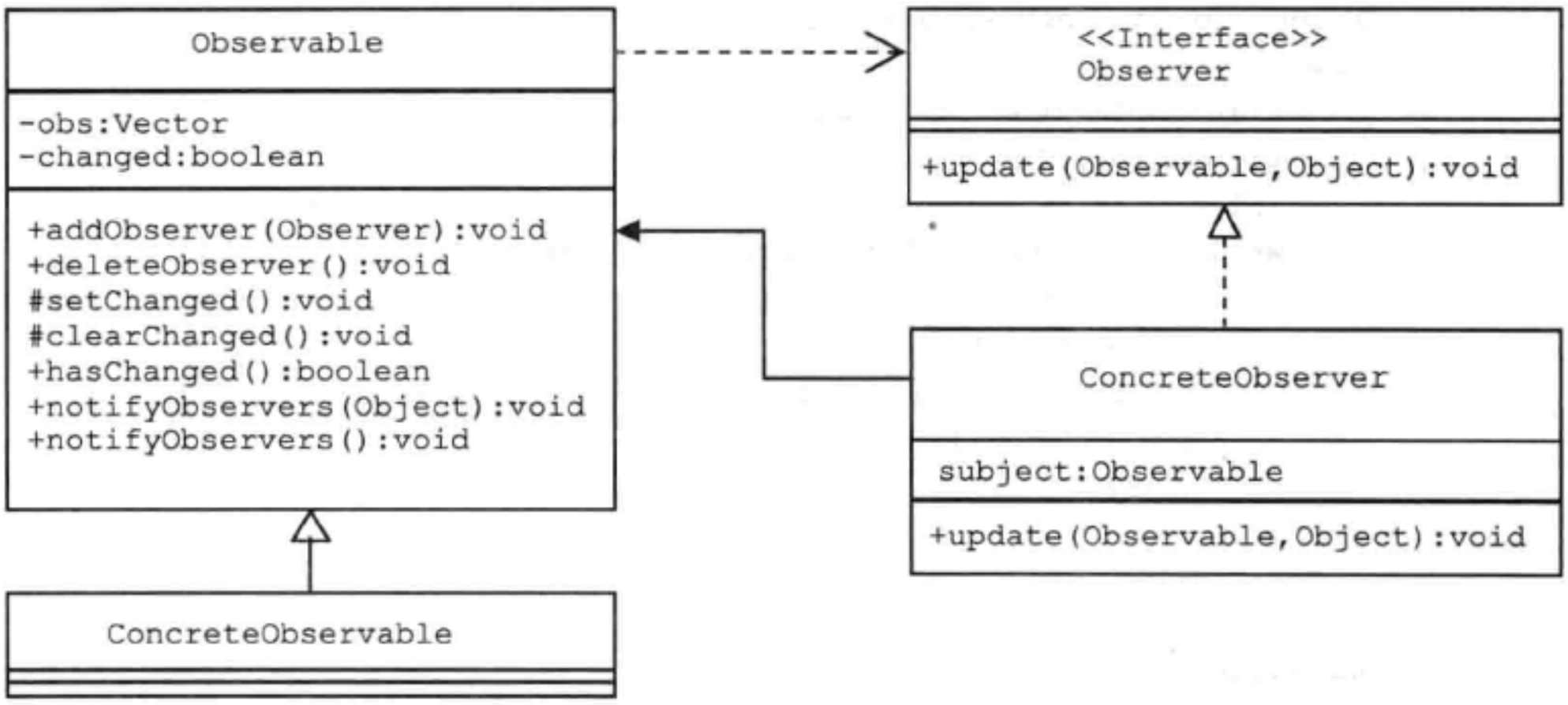


图 5-3　JDK 系统观察者设计模式类图

1. 利用 JDK 中的 Observer、Observable 完成观察者模式

下面重新编写 5.2 节的内容，步骤如下。

（1）编制主体类 Subject。

```
public class Subject extends java.util.Observable {
    String data;
    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;              //更新数据
        setChanged();                  //置更新数据标志
        notifyObservers(null);         //通知各个具体观察者
    }
}
```

无需编制自定义主体接口，直接从 Observable 类派生即可，在该类中主要定义各中心数据及 getter、setter 方法。getter 方法主要用于具体观察者“拉”数据，setter 方法主要用于更新数据、设置 changed 变量及通知各具体观察者进行数据响应。

（2）编制具体观察者类 OneObserver。

```
public class OneObserver implements java.util.Observer {
    public void update(Observable arg0, Object arg1) {
        Subject subject = (Subject)arg0;
        System.out.println("The data is:" +subject.getData()); //“拉”数据
    }
}
```

无需编制自定义观察者接口，直接实现 Observer 接口即可。Update()方法中主要完成“拉”数据及处理过程。

（3）一个简单的测试类。

```
public class Test {
```

```
    public static void main(String[] args)throws Exception {
        Observer obj = new OneObserver();  //定义观察者
        Subject s = new Subject();         //定义主题
        s.addObserver(obj);                //主题添加观察者
        s.setData("hello");                //主题更新数据
    }
}
```

利用 JDK 的系统类及接口，大大简化了观察者设计模式的程序编码。那是否表明 5.4 节之前讲的内容就无效了呢？不是。这些知识从最底层的接口讲起直至最高层，对于理解观察者模式的本质是必要的。值得注意的是，JAVA API 给出的 Observable 是一个类，不是一个接口。尽管该类为它的子类提供了很多可直接使用的方法，但也有一个问题：Observable 的子类无法使用继承方式复用其他类的方法，其原因是 Java 不支持多继承。这种情况下，如果用 5.2 节自定义主题接口 ISubject 就可以轻易实现。而且，信息世界飞速发展，需求分析千变万化，或许需要不同的 addObserver()、deleteObserver()、notifyObservers()方法，或许还要增加一些方法。也就是说，要定义更复杂的自定义主题或观察者接口，编制更复杂的主题类，这时就会发现直接用 Observable、Observer 或许很难解决现实问题。若熟知观察者模式的本质，则会方便实现。

2. Observerable 类和 Observer 接口代码分析

Observerable 类、Observer 接口均是专家级的代码，可以从中吸取许多好的经验。笔者认为主要有两点。

（1）设置标识变量。

主要体现在 changed 成员变量的设置。notifyObservers()是非常重要的一个方法，其 JDK 源码如下。

```
public void notifyObservers(Object arg) {
    Object[] arrLocal;
    synchronized (this) {
        if (!changed)
            return;
        arrLocal = obs.toArray();
        clearChanged();
    }
    for (int i = arrLocal.length-1; i>=0; i--)
        ((Observer)arrLocal[i]).update(this, arg);
}
```

可以看出，只有当 changed 为 true 时，观察者对象才能数据响应。而 5.2 节中只要调用 Subject 类中的 notifyObservers()方法，即使在中心数据没有刷新的情况下，观察者对象也能数据响应。经过对比，可以得出：若某方法非常关键，一定要考虑它有几种状态，从而定义标识变量来予以控制。

（2）形参的设定。

这里的形参主要指 Observable 类中的 notifyObservers(Object arg)方法参数和 Observer 接口中定义的 update(Observable,Object arg)方法中第二个形式参数。有了 arg 参数对象，可以把一些比较信息由主题动态传递给观察者，使编程更加灵活。

例如下面示例中，有两个观察者，一个负责统计满足“data=arg”出现的次数（arg 是动态传入的字符串），另一个在屏幕上显示 data 字符串。程序代码如下。

```
//主题类 Subject
public class Subject extends java.util.Observable {
    String data;
    Object factor; //增加条件变量
    public void setFactor(Object factor){
        this.factor = factor;
    }
    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;
        setChanged();
        notifyObservers(factor);//把条件也传递给观察者
    }
}
//第一个具体观察者类 OneObserver: 判断出现满足条件 factor 的元素个数
public class OneObserver implements Observer {
    private int c = 0;
    public int getC(){
        return c;
    }
    public void update(Observable obj, Object factor) {//此观察者用到了条件 factor
        Subject subject = (Subject)obj;
        if(subject.getData().equals((String)factor)) //判断“拉”数据是否满足条件 factor
            c ++;
    }
}
//第二个具体观察者类 TwoObserver: 屏幕输出元素
public class TwoObserver implements Observer {
    public void update(Observable obj, Object factor) { //此观察者没有用到条件 factor
        Subject subject = (Subject)obj;
        System.out.println("The data is:" +subject.getData());
    }
}
//一个简单测试类
public class Test {
    public static void main(String[] args)throws Exception {
        Subject s = new Subject();
        Observer obj = new OneObserver(); Observer obj2 = new TwoObserver();
        s.addObserver(obj); s.addObserver(obj2);
        s.setFactor("hello");      //为主题设置条件
        s.setData("hello"); s.setData("how are you");
        s.setData("hello"); s.setData("thanks");
        System.out.println("The hello times is:" +((OneObserver)obj).getC());//显示
出现 hello 字符串的次数
    }
}
```

研究源码能使我们受益非浅，弥补不足。但笔者也发现了一个小小的纰漏，那就是 Observable 与 Observer 在名字上太像了，英语语义也完全相近，但实际上，它们的语义差别很大，一个表示主题，另一个表示观察者。这只是笔者的一个观点，与大家共忖。因此，在

实际工程中一定要注意起名字，既要表义，又要好记。这不是小问题，却往往是许多程序工作者忽略的问题。因为应用系统一旦商业化或成为标准，就很难再修改名字了。

5.5　应 用 探 究

【例 5-1】 机房温度监测仿真功能。

为了方便说明问题，假设仅监测一个机房的温度数据。要求：①定间隔采集温度数值；②记录采集的温度数值；③标识异常的温度数值；④当温度值连续超过比较值 n 次时，要发报警信息。

分析：监测功能是以温度为中心的，因此用观察者设计模式实现程序架构是比较方便的。总体思想是：温度作为主题类，两个观察者类，一个观察者负责记录数据，另一个观察者负责异常处理。两个基本的策略是：① 将时间采样间隔数值、温度异常数值、连续温度越界极限数值、EMAIL 地址封装在 XML 配置文件 info.xml 中，EMAIL 是报警信息邮件。② 有单独的数据库表 normal 存储温度记录，但不通过在该表设置状态字段表明数值是否异常，设计单独的异常记录表 abnormal。这样做的好处是：abnormal 表的记录远比 normal 表的记录少得多，将来查询各种异常记录信息会非常快。③ 数据产生器采用反射技术。各种基本信息描述如表 5-2 所示。

程序采用控制台程序，用 JDK 的 Observable 类及 Observer 接口实现即可。

表 5-2　　机房温度监测仿真功能基本信息说明

Info.XML 配置文件

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM
 "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>Observer</comment>
  <entry key="range">2</entry>
  <entry key="limit">30</entry>
  <entry key="nums">5</entry>
  <entry key=”address”>aaa@163.com</entry>
  <entry key=”reflect”>DataRandom</entry>
</properties>
```

说　明

采样间隔 2 s，温度预警值 30℃。若连续采样 5 次温度值均超过预警值，则向 aaa@163.com 发送报警信息。由于现在邮箱有与电话绑定功能，则责任人就会及时获得预警信息。

反射的类名是 DataRandom

normal 记录数据表结构

序号	字段名	字段描述
1	wenduvalue	采集温度数值
2	recordtime	记录时间

说　明

abnormal 记录数据表结构

序号	字段名	字段描述
1	abnormalvalue	采集温度数值
2	recordtime	记录时间

说　明

各部分关键代码如下。

（1）条件类 Factor：

用于主体向观察者传送条件对象，包括温度预警值、预警极限次数值、预警邮箱等信息。

```
public class Factor {
    private int limit;          //温度预警值
    private int times;          //连续越过预警值次数极限值
    private String address;     //邮件地址
    public Factor(int limit,int nums,String address){
        this.limit=limit;this.times=nums;this.address=address;
    }
    public int getLimit() {
        return limit;
    }
    public int getTimes() {
        return times;
    }
    public String getAddress() {
        return address;
    }
}
```

（2）主题类 Subject。

```
public class Subject extends java.util.Observable {
    private int data;
    private Factor factor;
    public void setFactor(Factor factor){ //设置条件对象
        this.factor = factor;
    }
    public int getData() {
        return data;
    }
    public void setData(int data) {
        this.data = data;
        setChanged();                   //observable类中的方法
        notifyObservers(factor);        //将条件对象广播给各观察者
    }
}
```

（3）数据记录观察者类 DataObserver。

```
public class DataObserver implements Observer {
    public void update(Observable obj, Object factor) {
        Subject subject = (Subject)obj;
        String strSQL = "insert into normal values(" +subject.getData()+ ",now())";
        DbProc dbobj = new DbProc();
        try{
            dbobj.connect();
            dbobj.executeUpdate(strSQL);
            dbobj.close();
        }
        catch(Exception e){ }
    }
}
```

该类的功能是将采集到的所有数据保存到 normal 表中。

（4）异常数据观察者类 AbnormalObserver。

```
public class AbnormalObserver implements Observer {
    private int c = 0; //温度异常值累积
    public void update(Observable obj, Object factor) {
        Subject subject = (Subject)obj;
        Factor fac = (Factor)factor;
        if(subject.getData()<fac.getLimit()){   //若采集温度值<条件温度预警值
            c = 0;                              //则重新开始累积，返回
            return;
        }
        c ++;
        saveToAbnormal(subject);                //将越界数据保存至异常数据表
        if(c == fac.getTimes()){                //如果越界累积次数=条件极限次数
            sendEmail(fac);                     //则发送邮件
            c = 0;                              //重新开始累积
        }
    }
    private void saveToAbnormal(Subject subject){
        String strSQL = "insert into abnormal values(" +subject.getData()+ ",now())";
        DbProc dbobj = new DbProc();
        try{
            dbobj.connect();
            dbobj.executeUpdate(strSQL);
            dbobj.close();
        }
        catch(Exception e){ }
    }
    private void sendEmail(Factor factor){
        String host = "smtp.163.com";           //邮件服务器
        String from = "dqjbd@163.com";          //发件人地址
        String to = factor.getAddress();        //接受地址（必须支持 POP3 协议）
        String userName = "dqjbd";              //发件人用户名称
        String pwd = "123456";                  //发件人邮件密码
        Properties props = new Properties();
        props.put("mail.smtp.host", host);
        props.put("mail.smtp.auth", "true");

Session session = Session.getDefaultInstance(props);
        session.setDebug(true);

        MimeMessage msg = new MimeMessage(session);
    try {
        msg.setFrom(new InternetAddress(from));
        msg.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
        msg.setSubject("温度预警信息.");        //邮件主题
        msg.setText("机房温度处于异常状态");    //邮件内容
        msg.saveChanges();

        Transport transport = session.getTransport("smtp");
        transport.connect(host, userName, pwd);//邮件服务器验证
        transport.sendMessage(msg, msg.getRecipients(Message.RecipientType.TO));
```

```
        } catch (Exception e) {    }
    }
}
```

关键是理解 update()方法内的过程，说明如下。

过程　update(Observable, Object)：

① 如果获取的“拉”数据温度值 < 温度预警值

② 则重新置温度越界累积次数 c<—0, 返回

③ 累积次数 c<—c+1

④ 将越界温度值保存至异常数据库 abnormal 表中

⑤ 如果越界累积次数=极限次数

⑥ 则发送预警邮件

⑦ 重新置温度越界累积次数 c<—0, 返回

另外，发送邮件 sendEmail()方法采用了 Java Mail 技术。Java Mail API 是 SUN 公司为 Java 开发者提供的公用 Mail API 框架，它支持各种电子邮件通信协议，如 IMAP、POP3 和 SMTP，为 Java 应用程序提供了电子邮件处理的公共接口。其所需压缩库文件可在官网上下载。

（5）仿真数据生成器。

定义数据生成器类均从 ISimuData 自定义接口派生。借鉴 RecordSet 接口，ISimuData 接口定义如下。

```
public interface ISimuData<T>{
    void open();              //打开文件或其他
    void close();             //关闭文件或其他
    boolean hasNext();        //有下一条记录否?
    T next();                 //返回当前记录
}
```

当前正在采用的数据生成器类名信息保存在表 5-2 所述的 info.xml 配置文件中，类名是 DataRandom。它是 ISimuData 的子类，生成数据的方法多种多样。可从文件读取，或用随机算法产生，或者其他方式。这里就不详细列举该类的具体代码了，读者可以自行思考。

（6）利用反射机制构建主方法。

```
public class Test {
    public static void main(String[] args)throws Exception {
        FileInputStream in= new FileInputStream("info.xml");     //读配置文件
        Properties p = new Properties();
        p.loadFromXML(in);
        int range = Integer.parseInt(p.getProperty("range"));    //获得采集间隔
        String reflectClassName = p.getProperty("reflect");      //获得反射机制类名
        int limit = Integer.parseInt(p.getProperty("limit"));    //获得预警值
        int nums  = Integer.parseInt(p.getProperty("nums"));//获得连续温度越界极限次数
        String address = p.getProperty("address");               //获得电子邮件地址
        Factor factor = new Factor(limit,nums,address);
        in.close();

        Subject s = new Subject();//主题-观察者模式设计
        Observer obj = new DataObserver();
        Observer obj2 = new AbnormalObserver();
```

```
s.addObserver(obj); s.addObserver(obj2);
s.setFactor(factor);          //主题设置条件对象，以备广播给观察者对象用
//利用反射技术数据仿真
ISimuData<Integer> sdobj =
(ISimuData)Class.forName(reflectClassName).newInstance();
sdobj.open();
while(sdobj.hasNext()){
     int value = sdobj.next();
     s.setData(value);
     try{
          Thread.sleep(range*1000);
          }
          catch(Exception e){ }
     }
     sdobj.close();
  }
}
```

【例 5-2】 数据库表解析程序。

为了更好地理解程序架构，给出界面示例如图 5-4、图 5-5 所示。

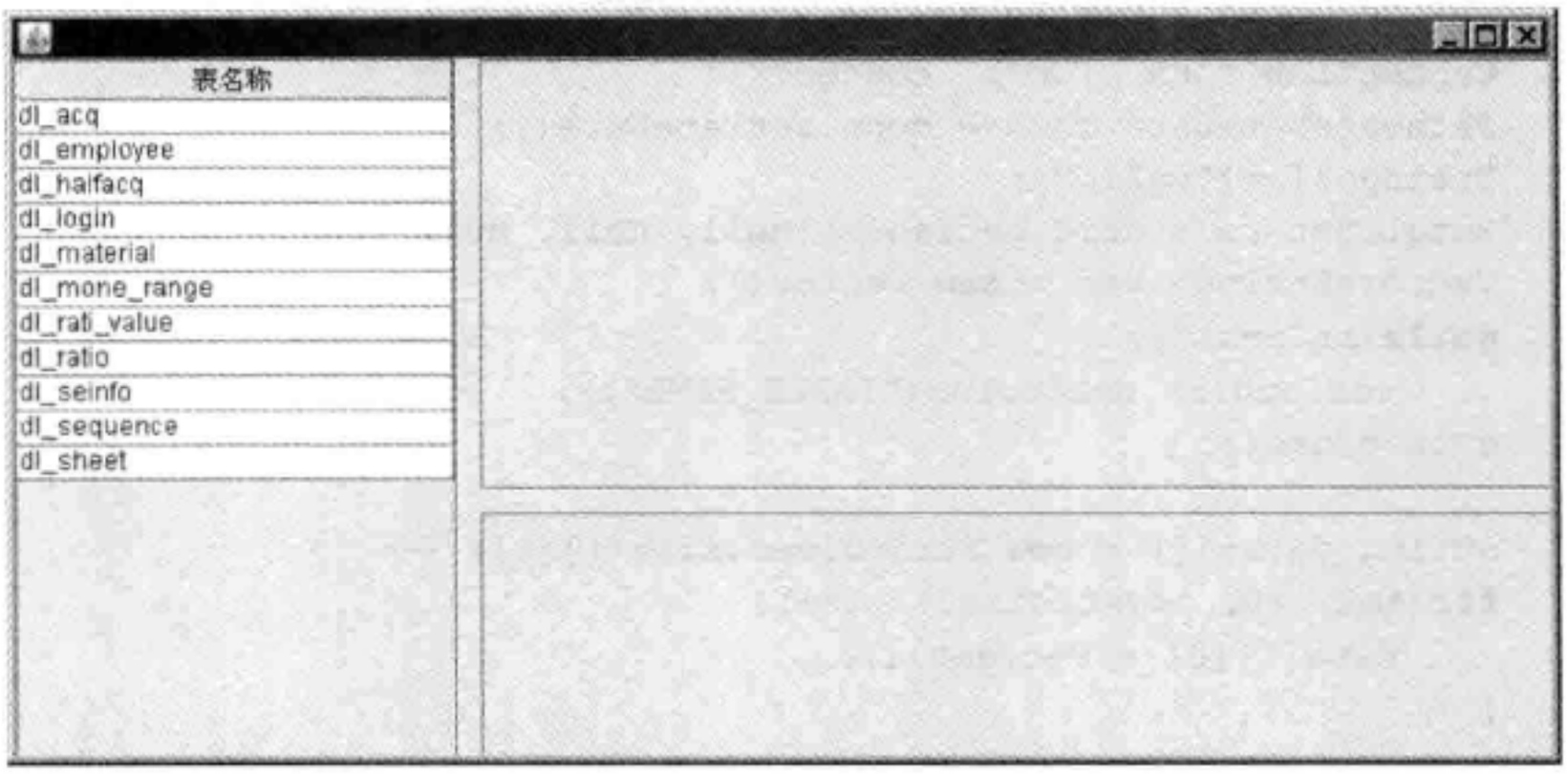


图 5-4　表解析程序初始界面

当鼠标选中某表后，响应界面如图 5-5 所示。

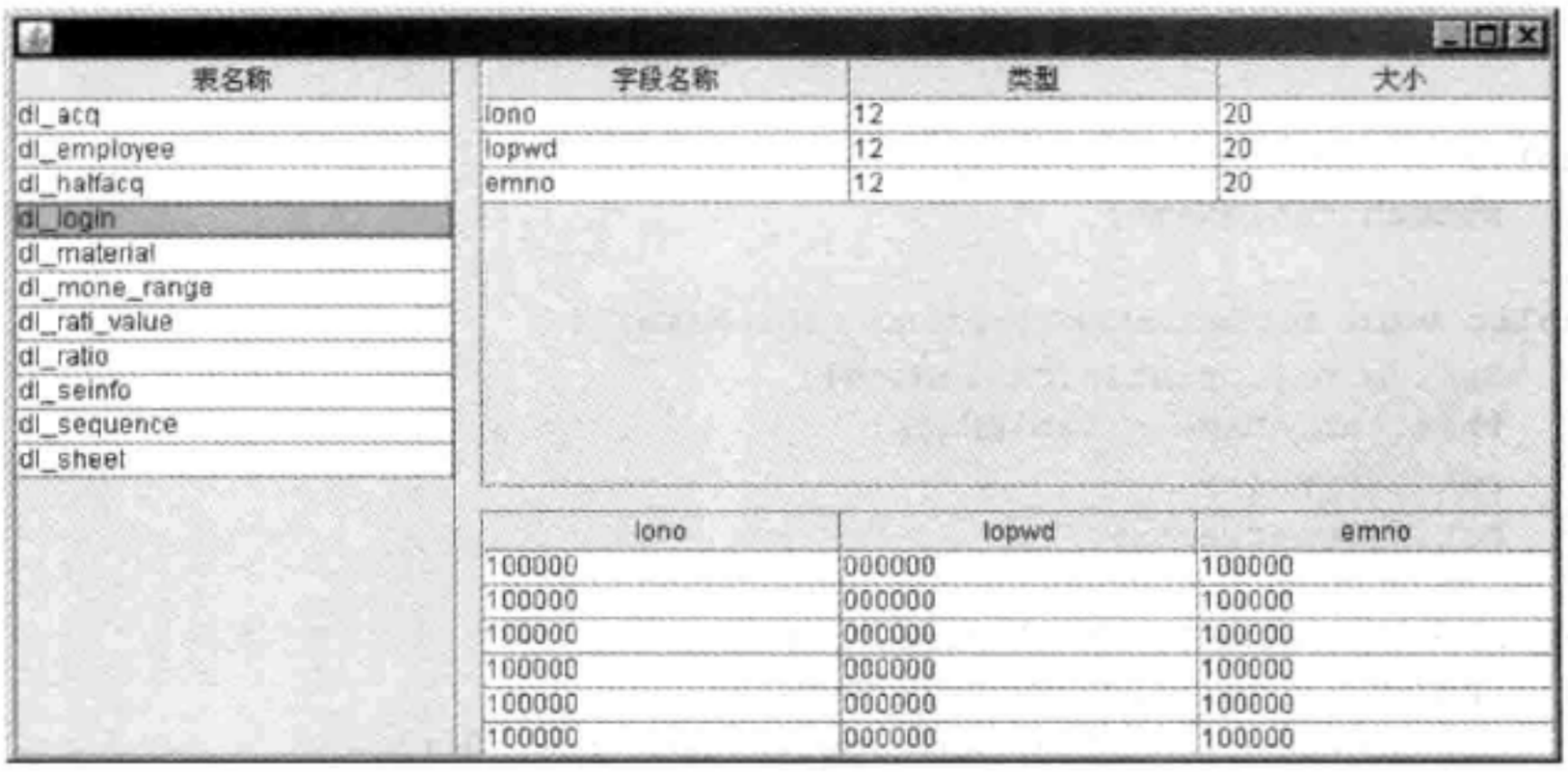


图 5-5　表解析响应界面

初始界面显示数据库中包含的表，响应界面中上方显示选中表的设计结构，下方显示表的数据信息。表结构及数据随着所选表不同而不同，因此可以用观察者模式完成所述功能。

之前讲述的观察者设计模式内容都是基于控制台的，可能有读者认为本题是基于图形用户界面的，难度似乎一下增加了许多，甚至很多读者不知如何定义主题类和观察者类。这主要是因为图形用户界面涉及界面生成及消息响应问题。解决这种问题最根本的方法是从语义出发，先完成观察者模式的功能类，再思考界面生成问题。本题仍采用 JDK 中的 Observable 及 Observer 共同完成。

（1）主题类 Subject。

从图 5-4 得出以下主要因素：①主题类与 JTable 有关；②主题类中心数据是表名称；③JTable 表格必须支持鼠标事件，因此该类必须实现 MouseListener 接口。关键代码如下。

```
public class Subject extends Observable implements MouseListener {
    private String tableName;    //主题中心数据是表名称
    private JTable table;        //主题与 JTable 相关
    public Subject(JTable table)throws Exception{
        this.table = table;

        DbProc dbobj = new DbProc();     //数据库工具类
        Connection conn = dbobj.connect();
        DatabaseMetaData dbmd = conn.getMetaData();
        String s[]={"table"};
        ResultSet rs = dbmd.getTables(null, null, null,s);
        Vector<String> vec = new Vector();
        while(rs.next())
            vec.add(rs.getString("TABLE_NAME"));
        conn.close();

        String data[][] = new String[vec.size()][1];
        for(int i=0; i<vec.size(); i++){
            data[i][0] = vec.get(i);
        }

        String title[] = {"表名称"};
        DefaultTableModel dtm = (DefaultTableModel)table.getModel();
        dtm.setDataVector(data, title);

        table.addMouseListener(this);  //主题类必须注册鼠标事件
}
public String getTableName() {
    return tableName;
}
public void setTableName(String tableName) {
    System.out.println(tableName);
    this.tableName = tableName;
    setChanged();
    notifyObservers();
}
public void mouseClicked(MouseEvent arg0) {
    int row = table.getSelectedRow();
    setTableName((String)table.getValueAt(row, 0));
}
```

```
        //其他 MouseListener 接口中定义的函数必须在此实现，在此略
    }
```

① 上面程序中,获得数据库表名称是由 java.sql 包中的 DatabaseMetaData 类的 getTables() 方法实现的，原型如下。

```
    ResultSet DatabaseMetaData.getTables(String catalog,  String schemaPattern, String
tableNamePattern,   String types[]) throws SQLException
```

参数解释如下。

- catalog：数据库目录名称，可设为 null。
- schemaPattern：方案名称的样式，可设为 null。
- tableNamePattern：表名称的样式，可以包含匹配符，如"TEST%"。
- types：要包括的表类型组成的列表，可设为 null，表示所有的。types 的常量值包括 "TABLE","VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM"。

② 填充 JTable 表格必须形成表头数组 title[]及二维数据数组 data[][]，然后获得 DefaultTableModel 默认模型类对象，最后调用 setDataVector()方法完成表格的填充，关键代码如下。即将讲到的具体观察者表格填充也是同样的思路。

```
DefaultTableModel dtm = (DefaultTableModel)table.getModel();
dtm.setDataVector(data, title);
```

（2）表结构观察者类。

从图 5-5 得出以下主要因素:①表结构观察者类与 JTable 有关；②JTable 表格仅是完成表结构功能，因此无须添加消息映射。关键代码如下。

```
public class StructObserver implements Observer {
    private JTable table;   //表结构观察者类与 JTable 有关
    public StructObserver(JTable table){
        this.table = table;
    }
    public void update(Observable obj, Object arg1) {
        Subject subject = (Subject)obj;
        String strSQL = "select * from " +subject.getTableName();
        DbProc dbobj = new DbProc();
        try{
            Connection conn= dbobj.connect();
            Statement stm = conn.createStatement();
            ResultSet rst = stm.executeQuery(strSQL);
            ResultSetMetaData rsmd = rst.getMetaData();
            String title[] = {"字段名称","类型","大小"};
            String data[][]= new String[rsmd.getColumnCount()][title.length];
            for(int i=0; i<rsmd.getColumnCount(); i++){
                data[i][0] = rsmd.getColumnName(i+1);
                data[i][1] = "" +rsmd.getColumnType(i+1);
                data[i][2] = "" +rsmd.getPrecision(i+1);
            }
            stm.close();
            conn.close();

            DefaultTableModel dtm = (DefaultTableModel)table.getModel();
            dtm.setDataVector(data, title);
```

```
            }
            catch(Exception e){}
        }
    }
```

观察者程序中，解析表结构主要是通过 java.sql 包中的 ResultSetMetaData 类完成的，其常用方法如下。

- int getColumnCount(): 返回列字段的数目。
- String getColumnName(int col): 根据字段的索引值（>=1）取得字段的名称。
- int getColumnType(int col): 根据字段的索引值（>=1）取得字段的类型。返回值定义在 java.sql.Type 类中。由于本例中直接显示了返回的整数值，故不能表义看出本字段的具体类型。
- int getCatalogName(int col): 获取指定列的表目录名称。
- int getColumnDisplaySize(int col): 获取指定列的最大标准宽度，以字符为单位。

（3）表数据显示观察者类 ShowObserver。

从图 5-5 得出以下主要因素：①表数据显示观察者类与 JTable 有关；②JTable 表格仅是完成表结构功能，因此无须添加消息映射。关键代码如下。

```
public class ShowObserver implements Observer {
    private JTable table; //表数据显示观察者类与 JTable 有关
    public ShowObserver(JTable table){
        this.table = table;
    }
    public void update(Observable obj, Object arg1) {
    Subject subject = (Subject)obj;
    String strSQL = "select * from " +subject.getTableName();
    DbProc dbobj = new DbProc();
    try{
        Connection conn= dbobj.connect();
        Statement stm = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
        ResultSet rst = stm.executeQuery(strSQL);
        rst.last();                          //游标指向最后一条记录
        int rows = rst.getRow();             //获得记录总数
        ResultSetMetaData rsmd = rst.getMetaData();

        //获得查询的列名称信息，保存到 title 数组中
        String title[] = new String[rsmd.getColumnCount()];
        String data[][]= new String[rows][title.length];
        for(int i=0; i<rsmd.getColumnCount(); i++){
            title[i] = rsmd.getColumnName(i+1);
        }

        //获得查询的二维记录集 data[][]
        rst.first();
        for(int i=0; i<rows; i++ ){
            for(int j=0; j<title.length; j++)
                data[i][j] = rst.getString(j+1);
        }
        stm.close();
        conn.close();
```

```
                DefaultTableModel dtm = (DefaultTableModel)table.getModel();
                dtm.setDataVector(data, title);
            }
            catch(Exception e){}
        }
    }
```

对不同表而言，查询结果的列字段数目、标题信息内容都是不同的。因此本部分主要实现了通用显示功能，关键思路是：首先获得查询记录集 rst，由 rst 获得 ResultSetMetaData 元数据对象 rsmd，根据 rsmd 获得列字段数目及标题数组 title[]；然后根据列信息及记录集行数可创建二维数据缓冲区 data[][]，遍历记录集 rst，形成真实 data[][]数据；最后完成 JTable 表格的填充。

（4）界面生成类 MyFrame。

本部分主要是形成界面及建立主题类与观察者类的关联，代码如下。

```
public class MyFrame extends JFrame {
    public void init(){
        //形成界面
        setLayout(null);

        JTable nameTable = new JTable();
        JTable structTable = new JTable();
        JTable showTable = new JTable();
        JScrollPane namePane = new JScrollPane(nameTable);
        JScrollPane structPane = new JScrollPane(structTable);
        JScrollPane showPane = new JScrollPane(showTable);

        JPanel left = new JPanel();left.setLayout(new BorderLayout());
        JPanel struct = new JPanel();struct.setLayout(new BorderLayout());
        JPanel show = new JPanel();show.setLayout(new BorderLayout());

        left.add(namePane);struct.add(structPane);show.add(showPane);
        add(left);add(struct);add(show);
        left.setSize(200, 500);
        struct.setSize(500,200);
        show.setSize(500, 290);
        left.setBounds(0, 0, left.getWidth(), left.getHeight());
        struct.setBounds(210,0,struct.getWidth(),struct.getHeight());
        show.setBounds(210, 210, show.getWidth(), show.getHeight());

        setSize(700,500);
        setResizable(false);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //设置主题-观察者之间的关联
        try{
            Subject subject = new Subject(nameTable);
            Observer obj = new StructObserver(structTable);
            Observer obj2= new ShowObserver(showTable);
            subject.addObserver(obj);
            subject.addObserver(obj2);
        }
```

```
            catch(Exception e){e.printStackTrace();}

        }

        public static void main(String[] args){
            new MyFrame().init();
        }
    }
```

可以看出，本示例主界面采用了 null 空布局，其他子面板只需计算好坐标，放在相应位置上就可以了。也可以说，某些时候设置为 null 布局，可能更方便形成主界面。

根据之前已编制的其他功能类分析，本部分无须加任何的消息响应。从这方面来看，在图形用户界面上应用设计模式，一定要从各子部分着手，千万不要把所有关联的东西都一起想，否则会适得其反。

第6章 桥接模式

6.1 问题的提出

有一类事物集合，设为 A1，A2，…，Am，每个事物都有功能 F1，F2，…，Fn。生活中有许多类似的现象。如邮局业务，如图 6-1 所示。

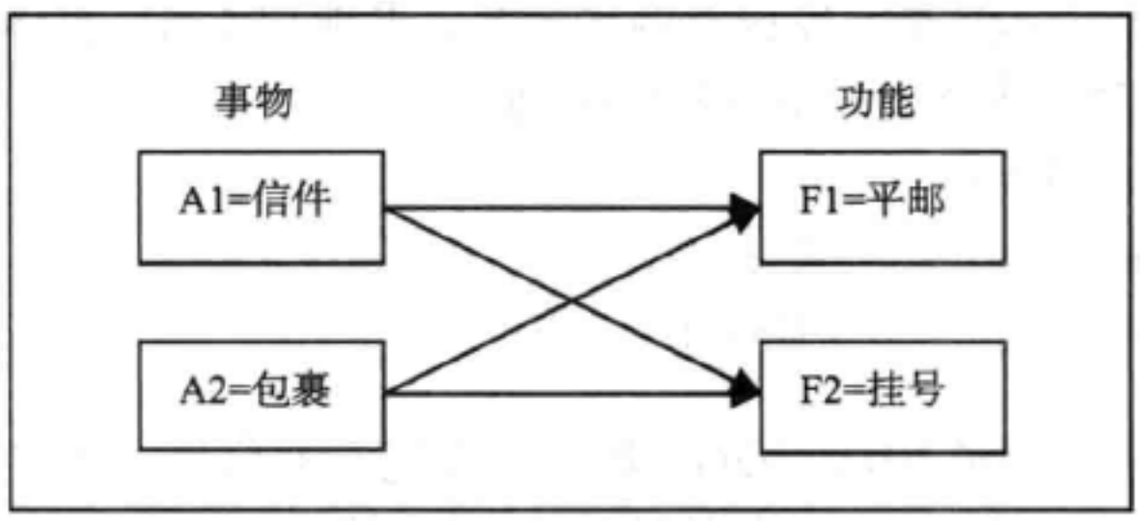


图 6-1 邮局业务示意图

图 6-1 描述的是邮局的发送业务。有两类事物：信件和包裹，均有平邮和挂号邮寄功能，那么用计算机如何来描述这些功能呢?

方法 1：

```
class A1F1{        //信件平邮
}
class A1F2{        //信件挂号
}
class A2F1{        //包裹平邮
}
class A2F2{        //包裹挂号
}
```

很明显，若有 m 个事物，n 个功能，按此方法，共要编制 $m \times n$ 个类。显然这是不科学的，一定是不可取的。

方法 2：

```
class A1{
    void F1(){}   //信件平邮
    void F2(){}   //信件挂号
}
class A2{
```

```
    void F1(){}   //包裹平邮
    void F2(){}   //包裹挂号
}
```

很明显，若有 m 个事物，n 个功能，按此方法，共要编制 m 个类。与方法 1 相比，编制类的数目减少了，但本质没有变，功能方法累积起来仍有 $m \times n$ 个。显然这同样是不科学的，也同样是不可取的。

那么，如何更好地解决图 6-1 所示的问题呢？桥接模式是重要的方法之一。

6.2 桥接模式

桥接模式是关于怎样将抽象部分与它的实现部分分离，使它们都可以独立地变化的成熟模式。

6.1 节中方法 1、方法 2 的根本缺陷是：在具体类中都封装了 F1()或 F2()方法。因此，一定有许多重复的代码。解决该问题的一个重要策略仍是利用语义，进一步抽象图 6-1 所示功能。可描述为：邮局有发送功能；发送有两种方式，平邮和挂号；具体事物可为信件和包裹。因此，主要把上述关键字转译成如下程序代码即可。

（1）定义邮寄接口 IPost。

```
public interface IPost{    //邮局
    public void post();    //发送功能
}
```

在 6.1 节中，一直用的是 F1()、F2()方法，没有把它抽象化。其实 F1()、F2()都是邮寄方法，所以是可进一步抽象的。IPost 相当于语义“邮局”，post()相当于“发送”。也许仍有许多读者认为：“你写出来，我就理解了；你没写出来，我就想不到”。造成这种情况的根本原因是把程序与生活实际割裂开来。其实，只要一想到“邮局有邮寄功能”，把它转译一下，不就是接口 IPost 吗？

（2）两个具体邮寄类 SimplePost、MarkPost。

```
//平信邮寄类 SimplePost
class SimplePost implements IPost{      //平信
    public void post(){                 //发送
        System.out.println("This is Simple post");
    }
}
//挂号邮寄类
class MarkPost implements IPost{        //挂号
    public void post(){                 //发送
        System.out.println("This is Mark post");
    }
}
```

经过（1）、（2）的论述，完成了语义的前半部分定义：邮局有发送功能；发送有两种方式，平邮和挂号。

（3）抽象事物类 AbstractThing。

```
abstract class AbstractThing{ //抽象事物
```

```
    private IPost obj;              //有抽象发送功能
    public AbstractThing(IPost obj){
        this.obj = obj;
    }
    public void post(){
        obj.post();
    }
}
```

该类是桥接模式的核心。分析语义“信件和包裹共享平邮与挂号功能”：信件、包裹是两个不同的事物，它们有共享的功能，也一定有相异的功能。共享的功能一定能封装到一个类中，又由于该类不能代表一个具体的事物，所以把它定义成 abstract 类是恰当的。

该类共享的是多态成员 obj，是 IPost 类型的，是抽象的、泛指的，用一条语句表明了事物共享平邮和发送功能。

（4）具体事物类 Letter、Parcel。

```
//信件类 Letter
class Letter extends AbstractThing{
    public Letter(IPost obj){
        super(obj);
    }
    //其他独有变量和方法
}
//包裹类 Parcel
class Parcel extends AbstractThing{
    public Parcel(IPost obj){
        super(obj);
    }
    //其他独有变量和方法
}
```

经过步骤（1）~（4），我们完成了桥接模式的编程过程，其 UML 如图 6-2 所示。

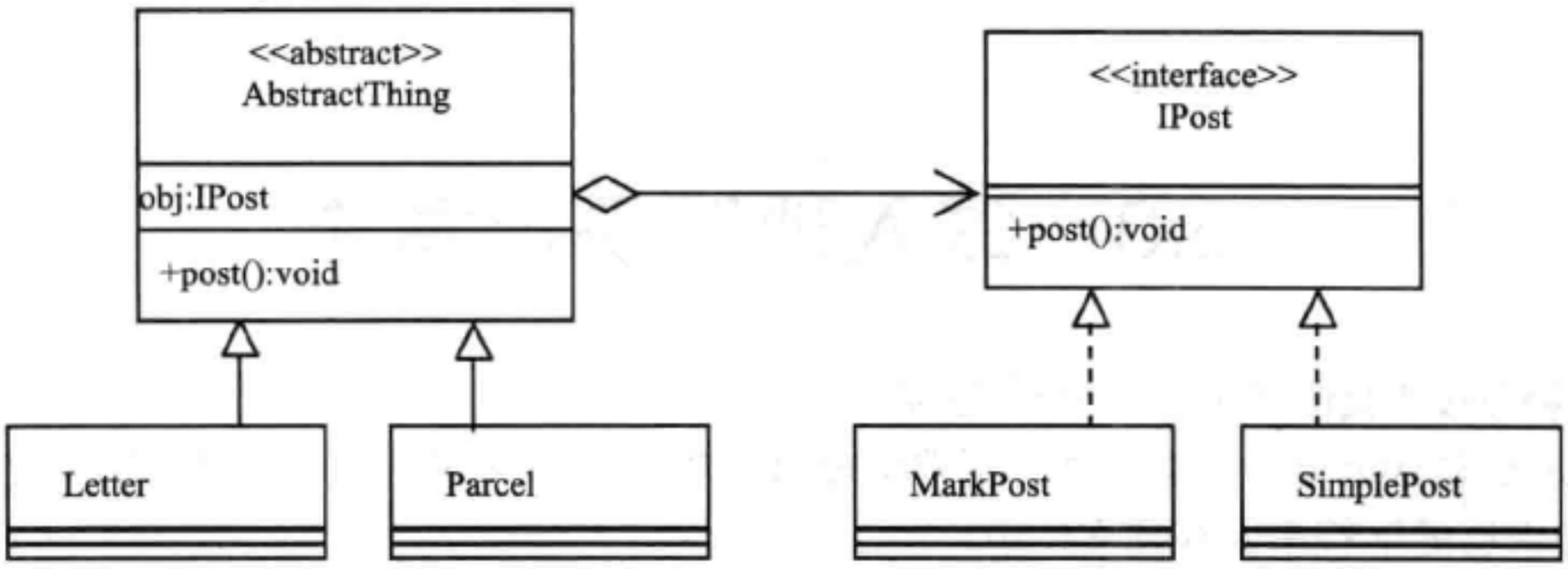


图 6-2 邮局发送过程桥接模式类图

通过 AbstractThing 类中的成员变量 obj，它就像桥梁一样，使得事物类与功能类巧妙地联系起来，这也是叫作桥接模式的一个重要原因。

现在，再编制一个简单的测试类，代码如下。

```
public class Test {
    public static void main(String[] args) {
        IPost p = new SimplePost();
```

```
            Letter letter = new Letter(p);
            letter.post();
        }
    }
```

结合图 6-2 得出该过程如下：先在右侧的功能类中选择一个具体的发送功能，再选择一个事物类，最后完成真正的发送过程。总结来说，桥接模式完成的是一个多条件选择问题。假设有二维条件，分别有 N1、N2 个选择，桥接模式要求在第 1 维 N1 个条件中选择一个，在第 2 维 N1 个条件中选择一个，最后完成有效组合。这其实与生活中的示例是相似的。例如有 10 件上衣、10 件裤子，一般来说，人们会分别在衣服中选一件，在裤子中选一件，而不会把衣服、裤子交叉在一起进行选择；再比如，人们要从大连坐火车去西安，从北京中转，一般来说会在大连到北京的 N 次火车中选一辆，再在北京到西安的 M 次火车中选一辆。生活中这些人们下意识地处理这类问题的方法其实与桥接模式算法是相似的。说到这里，或许你对桥接模式又有了更深刻的认识。

那么，现在思考一下，桥接模式结构是怎样满足需求分析的变化呢？前提是不论需求分析如何变化，都要满足图 6-1 所示功能图。主要有如下两种情况。

第一种情况：若增加了新的事物，则只需从 Abstract 派生一个类即可，其他无需改变。

```
class NewThing extends AbstractThing{
    public NewThing(IPost obj){
        super(obj);
    }
    //其他独有变量和方法
}
```

第二种情况：若增加了新的邮寄类别，比如特快专递，则只需从 IPost 接口派生一个类即可，其他无需改变。

```
class EMSPost implements IPost{          //特快专递
    public void post(){                  //发送
        System.out.println("This is EMS post");
    }
}
```

6.3 深入理解桥接模式

1. 桥接模式强调“包含”代替“继承”

日志是非常重要的一类文件，要求实现两种功能：① 将信息字符串直接保存到日志中；② 将加密后的字符串保存到文件中。

方法 1：

```
class LogFile{                           //将信息直接保存到日志文件
    public void save(String msg){
    }
}
class Encrypt extends LogFile{           //加密信息保存到文件
    public void save(String msg){
        msg = encrypt(msg);
        super.save(msg);
```

```
    }
    public String encrypt(String msg){
        String s = "";          //s是加密后的字符串
        //加密功能实现，略
        return s;
    }
}
```

方法2：

```
class LogFile{ //将信息直接保存到日志文件
    public void save(String msg){
    }
}
class Encrypt{ //加密信息保存到文件
    LogFile lf;
    public Encrypt(LogFile lf){
        this.lf = lf;
    }
    public void save(String msg){
        msg = encrypt(msg);
        lf.save(msg);
    }
    public String encrypt(String msg){
        String s = "";    //s是加密后的字符串
        //由于仅是论述程序架构，加密功能实现略
        return s;
    }
}
```

方法1中LogFile类、Encrypt类是派生关系，它的缺点是当LogFile父类改变时，有可能影响到Encrypt子类；方法2中LogFile类、Encrypt类是包含关系，Encrypt类中包含LogFile类型的成员变量，当LogFile类改变时，只要接口方法不改变，就不会影响到Encrypt类。方法2体现了桥接模式最基本的思想。本例虽然简单，没有定义任意的接口和抽象类，但对于理解桥接模式的本质是有帮助的，希望读者灵活运用。

2. 透过JDK理解桥接模式

JDK中有许多应用桥接模式的地方，例如Collections类中的sort()方法，源码如下。

```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
    Object[] a = list.toArray();
    Arrays.sort(a);
    ListIterator<T> i = list.listIterator();
    for (int j=0; j<a.length; j++) {
        i.next();
        i.set((T)a[j]);
    }
}
```

可以看出，集合对象排序是借助Arrays中的sort()方法完成的。从中也能分析出JDK设计人员的设计思想，即数组和集合类都有排序功能，必须实现数组排序算法；集合对象要先转换成数组，排序后，再填充集合对象即可。可以看出，集合排序结构类似桥接模式结构。

同理，集合随机序列产生方法 shuffle()等的实现也体现了桥接模式的思想。

其实，稍加改造，运用桥接思想，可以形成更高效的自定义集合类 MyList，包含排序、二分查找、随机序列生成等方法。读者也可在此基础之上进行扩充。代码如下。

```
class MyList<T>{
    List<T> list;
    Object[] arr;
    public MyList(List<T> vec){
        list = vec; arr = vec.toArray();    //在构造方法中直接将集合对象转为数组
    }
public void sort(){
    Arrays.sort(arr);                       //直接对数组派序
}
public int binarySearch(T key){
    int pos = Arrays.binarySearch(arr, key); //直接对排序后的数组二分查找
    return pos;
}
public void show(){         //当前数组显示
    for(int i=0; i<arr.length; i++){
        System.out.print(arr[i] + "\t");
    }
}
public void fill(){         //将数组填充回集合
    ListIterator<T> i = list.listIterator();
    for (int j=0; j<arr.length; j++) {
            i.next();
            i.set((T)arr[j]);
        }
    }
}
```

3. 反射与桥接模式

利用反射机制对 6.2 节邮局功能桥接模式代码进行修改，代码如下。

```
interface IPost{
    public void post();
}
abstract class AbstractThing{
    IPost obj;
    public AbstractThing(String reflectName)throws Exception{
        //利用反射机制加载功能类
        obj = (IPost)Class.forName(reflectName).newInstance();
    }
    public void post(){
        obj.post();
    }
}
class Letter extends AbstractThing{
    //属性和方法
}
//包裹类 Parcel
```

```
class Parcel extends AbstractThing{
    //属性和方法
}
```

可以看出，对功能类利用了反射技术，事物类中构造方法参数由 IPost 类型转化为字符串类型，表明反射机制要加载的功能类的类名。

进一步，如果对事物类、功能类均利用反射机制，如何实现呢？其实，只要传入事物类名、功能类名两个字符串就可以了。

下面是完整地利用反射机制实现桥接模式的代码。

```
interface IPost{
    public void post();
}
abstract class AbstractThing{
    IPost obj;
    public void createPost(String funcName) throws Exception{
    //利用反射机制加载功能类对象
    obj = (IPost)Class.forName(funcName).newInstance();
    }
    public void post(){
        obj.post();
    }
}
class ThingManage{    //事物管理类
    AbstractThing thing;
    AbstractThing createThing(String thingName) throws Exception{
    //利用反射机制加载事物类对象
        thing = (AbstractThing)Class.forName(thingName).newInstance();
        return thing;
    }
}
```

下面是本例涉及的具体事物类。

```
//信件类 Letter
class Letter extends AbstractThing {
    // 属性和方法
}
//平信邮寄类 SimplePost
class SimplePost implements IPost {                    // 平信
    public void post() {                               // 发送
        System.out.println("This is Simple post");
    }
}
```

与之前的代码相比较，增加了事物管理类 ThingManage，用以封装 AbstractThing。本部分代码核心思想是利用反射机制产生 AbstractThing 对象，在此基础之上，利用反射机制再产生 IPost 功能类对象，最后完成实际的事物发送功能。

一个简单的测试类代码如下。

```
public class Test {
    public static void main(String[] args)throws Exception{
        ThingManage obj = new ThingManage();
```

```
        AbstractThing thing = obj.createThing("Letter"); //创建事物类对象
        thing.createPost("SimplePost");                  //创建功能类对象
        thing.post();                                    //事物执行功能
    }
}
```

6.4 应 用 探 究

【例 6-1】 编制功能类，要求能读本地或远程 URL 文件，文件类型是文本文件或图像文件。

分析：很明显，该功能可由桥接模式完成。事物类指本地文件及 URL 文件类，功能类指读文本文件、读图像文件。本示例列举了两种实现方法，具体如下。

方法 1：

本方法仅列出了功能类框架。读者应着重理解该方法的缺陷，加深对桥接模式的理解。

（1）抽象功能类 AbstractRead。

```
abstract class AbstractRead<T>{
    public abstract T read(String strPath) throws Exception;
}
```

抽象方法 read()返回值是泛型类型，这是因为读文本文件应该返回字符串，而读图像文件一般来说只能返回二进制字节缓冲区。

（2）具体功能实现类。

```
//读文本文件类
class TextRead extends AbstractRead<String>{
    public String read(String strPath)throws Exception{
    return null;
    }
}
//读图像文件类
class ImgRead extends AbstractRead<byte[]>{
    public byte[] read(String strPath)throws Exception{
        return null;
    }
}
```

（3）抽象事物类。

```
class AbstractThing{        //抽象事物类
    AbstractRead reader;  //有抽象读功能
    public AbstractThing(AbstractRead read){
        this.reader = reader;
    }
    Object read()throws Exception{
        return reader.read();
    }
}
```

为什么 read()方法返回值是 Object 类型呢？因为读文本文件要求返回 String 类型，读图像文件要求返回 byte[]，为了屏蔽这种差异，返回值采用了 Object 类型。

（4）具体事物类。

包括两种事物类：本地文件及URL文件类，代码如下。

```
class NativeFile extends AbstractThing{
    public NativeFile(AbstractRead reader){
        super(reader);
    }
}
class URLFile extends AbstractThing{
    public URLFile(AbstractRead reader){
        super(reader);
    }
}
```

很明显，上述代码是按照桥接模式模型（与图6-2UML相似）直接转译过去的。其实仔细分析就会发现主要问题：在具体实现类TextRead、ImgRead中无法写代码了。以TextRead类为例，多态方法read()参数是文件路径strPath。常规思路是必须根据strPath获得字节输入流InputStream对象，但是对于本地文件、URL文件获得InputStream对象是不同的。本地文件获取InputStream对象的方法是：

```
InputStream in=new FileInputStream(strPath);
```

而URL文件获取InputStream对象的方法是：

```
URL u=new URL(strPath);
InputStream in=u.openStream();
```

可能有读者说，直接在多态方法read()中再加上一个标志位就可以了，使用下面的代码。

```
class TextRead extends AbstractRead<String>{
    public String read(String strPath, int type)throws Exception{
        InputStream in = null;
        switch(type){
            case 1:
                In = new FileInputStream(strPath);break;
            case 2:
                URL u = new URL(strPath);
                in = u.openStream();
        }
        //其他代码
    }
}
```

如果这样修改的话，那么也需要在ImageRead类中重写一遍同样的代码。夸张来说，如果有N个×××Read类，就要重写N次，很明显这是不科学的。

可能有读者说，本地读写文本、图像文件与URL读取文本、图像文件是不同的，那直接用4个类封装不就可以了吗？如下所示。

```
class TextRead extends AbstractRead<String>{…}        //本地读文本文件
class ImgRead extends AbstractRead<String>{…}         //本地读图像文件
class URLTextRead extends AbstractRead<String>{…}     //URL读文本文件
class URLImgRead  extends AbstractRead<String>{…}     //URL读图像文件
```

这就更不对了，理由见6.1节描述。

造成上述编码的根本原因在于对桥接模式的理解不够深入，主要有如下几点。

- 图6-2邮局功能示例只能使读者明晓桥接模式的大致功能，缺乏对细节的理解。若想

弄清楚细节，必须在非常具体的程序中亲自实践。

- 桥接模式 UML 类图中若具体事物类有 *M* 个，具体功能类有 *N* 个，则最终编制的程序中一般来说要有 *M* 个事物类、*N* 个功能类。一定要对功能类特别小心。如本例中已经明确说功能类是读文本文件、图像文件，那么意味着有两个功能类。表明的含义是：不论有多少个事物类，读文本文件是一致的，读图像文件是一致的。如果编出了上文所说的 TextRead、ImgRead、URLTextRead、URLImgRead 4 个类，那一定是不恰当的。
- 功能类的抽取至关重要，必须把多个事物类完成某个功能类的共性部分抽取出来，才能作为桥接模式的功能类,这也是理解桥接模式最重要的地方。而上文图 6-2 描述的邮局业务“平邮和挂号”对多数人来说并不熟知，不知道平邮、挂号有哪些相同、不同的地方，这就造成了无法确定功能类多态所需的参数序列，因此是很难体会到桥接模式的精华的。

方法 2：

关键思路是一定要弄清楚本地读文件、URL 读文件功能类有哪些异同点。读文件一般主要有三步：打开文件、读文件、关闭文件。打开文件是为了获得 InputStream 对象 in，只有这一步对本地文件和 URL 文件是不同的，有了 in 对象，后续的读文件、关闭文件都是相同的。

另外，如果缓冲区大小与文件大小是一致的,读文件的效率就高，因此必须获得文件长度值，这对本地、URL 文件的获取方法也是不一致的。

有了上述关键两点，编制的具体代码如下。

（1）抽象功能类。

```
interface IRead<T>{
    T read()throws Exception;
}
```

（2）具体功能实现类。

```
//读文本文件
class TextRead implements IRead<String>{
    AbstractStream stream;
    public TextRead(AbstractStream  stream){
        this.stream = stream;
    }
    public String read() throws Exception{
        byte buf[] = stream.readBytes();
        String s = new String(buf);
        return s;
    }
}
//读图像文件
class ImgRead implements IRead<byte[]>{
    AbstractStream stream;
    public ImgRead(AbstractStream  stream){
        this.stream = stream;
    }
    public byte[] read() throws Exception{
        return stream.readBytes();
    }
}
```

可以看出，内部封装了一个自定义流类 AbstractStream。它是动态变化的，可能指向本地

文件，也可能指向 URL 文件。AbstractStream 类及相关类的具体代码如下。

```
//抽象基类流
abstract class AbstractStream{
    protected InputStream in;
    protected int size;
    protected byte[] readBytes() throws Exception{
        byte buf[] = new byte[size];
        in.read(buf);
        return buf;
    }
    public void close() throws Exception{
        in.close();
    }
}
//指向本地文件流
class NativeStream extends AbstractStream{
    public NativeStream(String strFile) throws Exception{
        File f = new File(strFile);
        size = (int)f.length();
        in = new FileInputStream(f);
    }
    //其他代码
}
//指向 URL 文件流
class URLStream  extends AbstractStream{
    public URLStream(String strFile) throws Exception{
        URL url = new URL(strFile);
        in = url.openStream();
        HttpURLConnection urlcon = (HttpURLConnection)url.openConnection();
        size = urlcon.getContentLength();
    }
    //其他代码
}
```

关键思想是将流的共性封装在抽象类中，如读文件、关闭文件；将差异封装在子类中，如打开文件获得 InputStream 对象 in 及获得的文件长度 size。

（3）抽象事物类。

```
abstract class AbstractThing{
    IRead read;
    public AbstractThing(IRead read){
        this.read = read;
    }
    Object read() throws Exception{
        return read.read();
    }
}
```

（4）具体事物类。

```
//本地文件
class NativeFile extends AbstractThing{
    public NativeFile(IRead read){
        super(read);
    }
```

```
        //其他代码
    }
    //URL 文件
    class URLFile extends AbstractThing{
        public URLFile(IRead read){
            super(read);
        }
        //其他代码
    }
```

（5）一个简单的测试代码。

```
public class Test {
    public static void main(String[] args) throws Exception {
        //打开远程文件流
        AbstractStream in =
        new URLStream("http://localhost:8080/LnnuScience/login.jsp");
        TextRead textread = new TextRead(in);              //设置读文本文件
        AbstractThing thing = new URLFile(textread);       //设置读远程文件
        String s = (String)thing.read();                   //开始读远程文本文件过程
        in.close();
        System.out.println(s);
    }
}
```

总之，通过上述具体示例可知：桥接模式中功能类共性的抽取是非常关键的，而如何屏蔽具体功能的参数等差异，则是实现桥接模式的关键。如文中的 AbstractStream 类，希望读者一定要认真加以思考。

【例 6-2】 B/S 下数据显示功能。

假设 B/S 下数据显示通常有两种方式：table 表格及 JFreeChart 图表。例如显示一年中员工月工资及产品月销售金额，数据库中相应表已存在，如表 6-1 所示。

表 6-1　　数据库表说明

员工工资表-salary				
序号	字段名	类型	关键字	描述
1	emno	字符串	√	员工编号
2	emname	字符串		员工姓名
3	one，two，…，twelve	整型		代表 12 个月字段，是简化写法
产品月销售金额表-product				
1	prno	字符串	√	产品编号
2	prname	字符串		产品名称
3	one，two，…，twelve	整型		代表 12 个月字段，是简化写法

程序在 Tomcat 服务器中运行。当服务器启动后，直接在 IE 地址中输入地址：

```
http://localhost:8080/salaryservlet?emno=1000&type=1
```

其中，salaryservlet 为 URL，emno 为员工编号，type = 1 表示显示方式为表格，显示员工

编号为 1000 的工资信息，如图 6-3 所示。

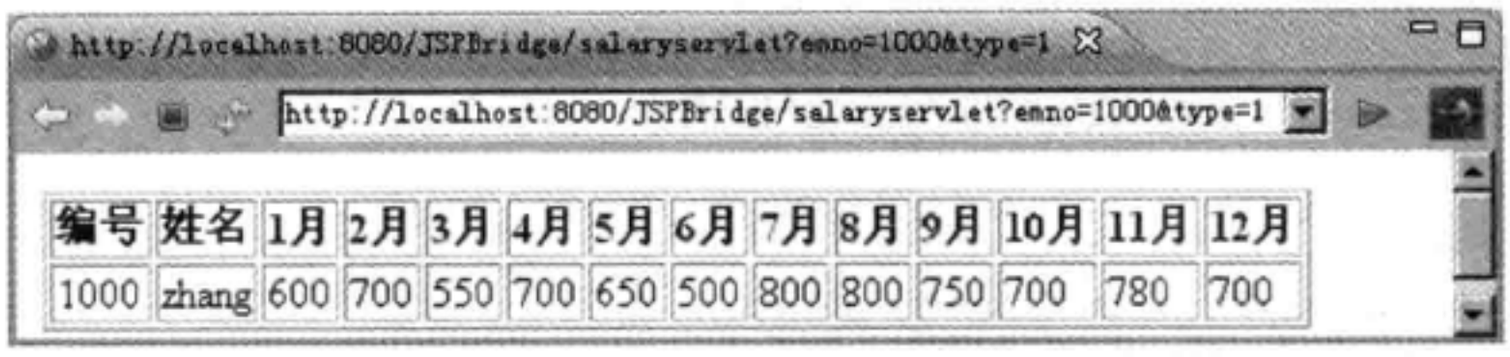


图 6-3 员工工资表格显示图

当输入地址 http://localhost:8080/ salaryservlet?emno=1000&type=2 时，显示为图 6-4 所示。

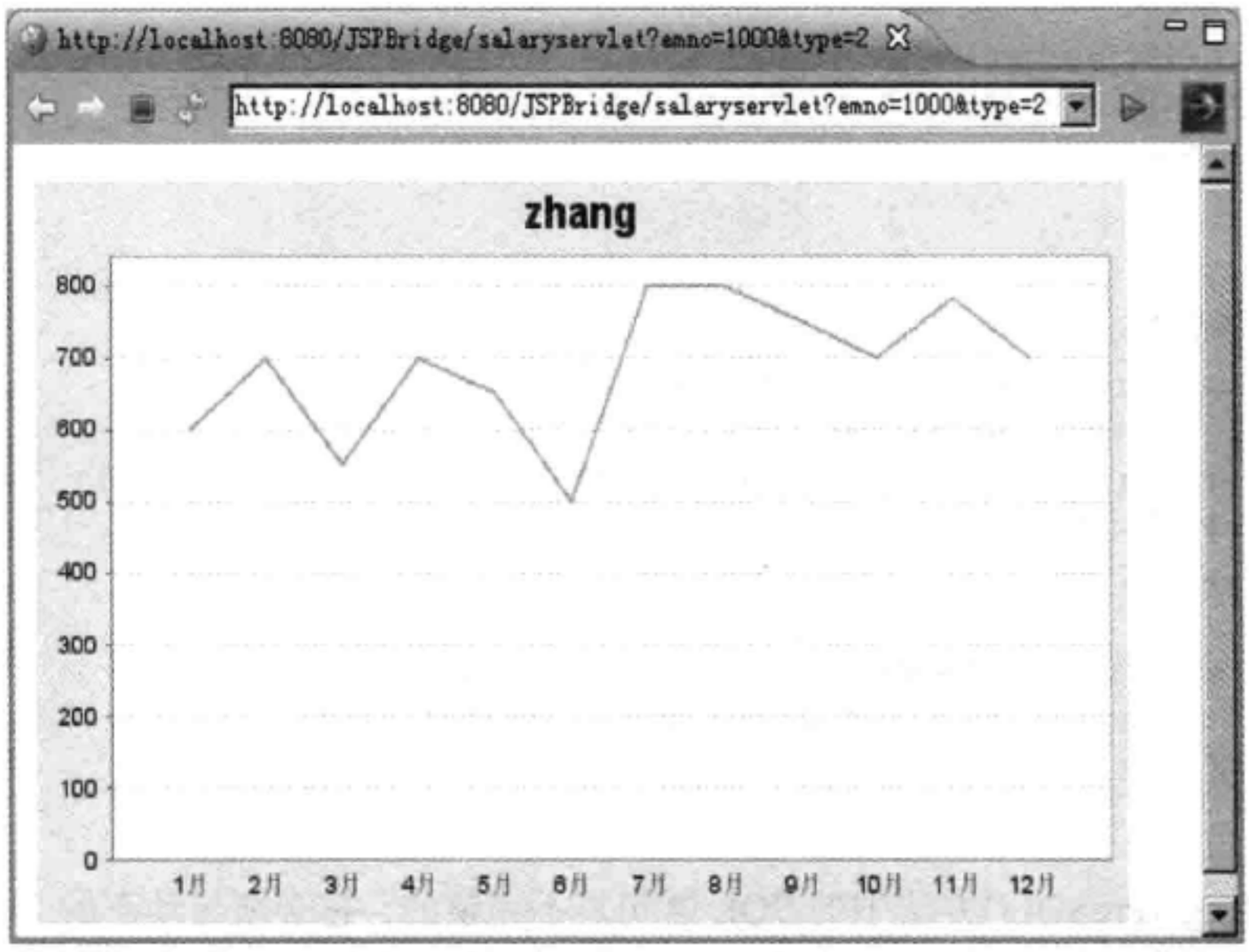


图 6-4 员工工资 JFreeChart 显示图

为了简化程序设计，假设数据有表格或 JFreeChart 显示功能，其字段与表 6-1 描述的相同，即第 1 个字段是“××编号”，第 2 个字段是“××名称”，后续的 N 个（可不确定）字段是整型数据。很明显，该类功能可由桥接模式实现，具体代码如下。

（1）抽象功能类及其参数。

```
public abstract class AbstractShow {
    public IPara para;
    public void setPara(IPara para){
        this.para = para;
    }
    abstract public String show(String no) throws Exception;
}
```

抽象方法 show()参数 no 代表“××编号”，要根据该值查询数据库表，获得符合条件数据。

IPara 是多态参数对象接口。在表格或 JFreeChart 中，除了显示数据信息外，还有一些辅助信息，如表头信息、JFreeChart 坐标说明参数、SQL 语句等。而这些辅助信息对员工或产品而言是不同的，因此必须加以屏蔽。IPara 接口及实现类相关具体定义如下。

```
    //参数接口定义
    public interface IPara {
        public String[] getTitle(); //表头信息不同
        public String getPreSQL();//SQL 语句不同
    }
    //员工具体参数定义
    public class SalaryPara implements IPara {
        public String[] getTitle() {
            String s[]={"编号","姓名","1 月","2 月","3 月","4 月","5 月","6 月","7 月","8 月","9
月","10 月","11 月","12 月"};
            return s;
        }
        public String getPreSQL() {
            String s = "select * from salary where emno=";
            return s;
        }

    }
    //产品具体参数定义
    public class ProductPara implements IPara {
        public String[] getTitle() {
            String s[]={"编号","产品名称","1 月","2 月","3 月","4 月","5 月","6 月","7 月","8
月","9 月","10 月","11 月","12 月"};
            return s;
        }
        public String getPreSQL() {
            String s = "select * from product where prno=";
            return s;
        }
    }
```

可以看出，getPreSQL()方法中的 SQL 语句是不完整的，它必须与动态的“××编号”合成才能形成实际的查询语句，该功能是在 AbstractShow 类中的 show()多态方法中完成的，后续还有说明。

（2）具体功能实现类。

```
//①表格功能实现类
public class TableShow extends AbstractShow {
    private String getHeader(){    //形成表头信息 HTML 字符串
        String title[] = para.getTitle();  //para 来自于抽象父类 AbstractShow
        String s = "<tr>";
        for(int i=0; i<title.length; i++){
            s += "<th>" +title[i]+ "</th>";
        }
        s += "</tr>";
        return s;
    }
    private String getData(String no)throws Exception{//形成数据显示 HTML 字符串
        String s = "";
        String strSQL = para.getPreSQL() +"'"+ no +"'"; //形成实际 SQL 语句
        DbProc dbobj = new DbProc();
        Connection conn = dbobj.connect();
```

```
Statement stm = conn.createStatement();
ResultSet rst = stm.executeQuery(strSQL);
if(rst.next()){
    s += "<tr>";
    for(int i=0; i<para.getTitle().length; i++){
        s += "<td>" +rst.getString(i+1)+ "</td>";
    }
    s += "</tr>";
}

stm.close();
        conn.close();
        return s;
    }
    public String show(String no)throws Exception {
        String s = "<table border='1'>";
        s += getHeader();      //添加表头 HTML 字符串
        s += getData(no);      //添加数据 HTML 字符串
        s += "</table>";
        return s;              //返回最终 HTML 字符串
    }
}

//②JFreeChart 图表显示功能类
public class GraphShow extends AbstractShow {
    public String show(String no) throws Exception{
        DefaultCategoryDataset dataset = new DefaultCategoryDataset();

        String s = "";
        String title[]=para.getTitle();
        String strSQL = para.getPreSQL() +"'"+ no +"'"; //形成实际 SQL 语句
        DbProc dbobj = new DbProc();
        Connection conn = dbobj.connect();
        Statement stm = conn.createStatement();
        ResultSet rst = stm.executeQuery(strSQL);
        String name = "";
        if(rst.next()){
            name = rst.getString(2);   //表中第 2 个字段是“员工姓名”或“产品名称”
            for(int i=3; i<=title.length; i++){
                dataset.addValue(rst.getInt(i), name, title[i-1]);
            }
        }
        JFreeChart chart = ChartFactory.createLineChart(name, "","",dataset,
                                PlotOrientation.VERTICAL,
                                false, false,false);
            ChartUtilities.saveChartAsJPEG(new File("d:/tmp.jpg"), 100, chart, 500,
                                        300);

            stm.close();
            conn.close();
            s = "<img src='d:/tmp.jpg' border='0'></img>";//形成<img>标签
            return s;
        }
    }
```

关键思想是利用 JFreeChart 填充数据，保存成 JPG 图像文件，并最终形成包含此图像的<img>标签字符串，返回给调用端。

JFreeChart 是开放源代码的 Java 图表生成组件。它使用 Java2D 接口进行开发，主要用来生成各种各样的图表，主要包括饼图、曲线图、柱状图、干特图等图形，生成 JPG、PNG 等格式的图片文件。

可从网站 http://sourceforge.net/projects/jfreechart 下载 JFreeChart 的最新版本。开发 JFreeChart 图表，需要两个类包支持：JFreeChart、JCommon，当前最新版本是 jfreechart-1.0.3.jar、jcommon-1.0.6.jar，解压 JFreeChart 后，在其 lib 文件夹中可以找到。

（3）抽象事物类。

```
public class AbstractThing {
    private AbstractShow show;
    public AbstractThing(AbstractShow show){
        this.show = show;
    }
    public String show(String no)throws Exception{
        return show.show(no);
    }
}
```

（4）具体事物类。

```
public class Employee extends AbstractThing {    //员工类
    public Employee(AbstractShow show){
        super(show);
    }
}
public class Product extends AbstractThing {    // 产品类
    public Product(AbstractShow show){
        super(show);
    }
}
```

（5）一个简单的 servlet 测试类。

建立显示员工工资的 servlet 类 SalaryServlet，URL 为 salaryservlet，其可从 request 中接收两个参数的值。一个是 emno，表明可获得员工编号；另一个是 type，表明可获得显示类型，type 为 1 表明需要表格显示，type 为 2 表明需要 JFreeChart 显示。代码如下。

```
public class SalaryServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public SalaryServlet() {
    }
    protected void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        response.setContentType("text/html; charset=utf-8");          //支持中文返回

        String emno = request.getParameter("emno");                   //获得员工编号
        int type = Integer.parseInt(request.getParameter("type"));    //获得显示类型
        IPara para = new SalaryPara();      //获得参数
        AbstractShow show = null;           //显示功能
        switch(type){
        case 1:
            show = new TableShow();break;  //表格显示
```

```
case 2:
    show = new GraphShow();break;                    //图表显示
}

show.setPara(para);                                  //为显示功能设置表头等参数
AbstractThing thing = new Employee(show);            //创建员工事物对象

String s = "";
try{
    s = thing.show(emno);                            //获得最终的 HTML 字符串
    }
    catch(Exception e){e.printStackTrace(); }
    response.getWriter().print(s);                   //返回给客户端
  }
}
```

程序 SalaryServlet 在浏览器运行时，需要在 IE 地址中输入地址:

http://localhost:8080/salaryservlet?emno=1000&type=1，通过浏览器地址栏向 Servlet 传递参数并运行。

第7章 代理模式

7.1 问题的提出

在生活中常遇到这样的事情：长虹电视的原产地在四川，你在大连工作，准备买长虹电视，那么你一定会在大连的商店或相关部门购买，而不会去四川购买；你去北京求职，一定会先到求职中心；你准备买车，一定会先去 4S 店；等等。可能你会说这类问题太司空见惯了。仔细分析会发现这类事件的一个重大特点，那就是客户都没有和“源产地”直接接触，而是通过一个称为“代理”的第三者来实现间接引用。代理对象可以在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不能看到的内容和服务或者添加客户需要的额外服务。代理模式则是一种可以很好实现客户对象与代理对象分离的策略。

7.2 代理模式

代理模式的定义如下：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫作 Proxy 或 Surrogate，它是一种对象结构型模式。其抽象 UML 图如图 7-1 所示。

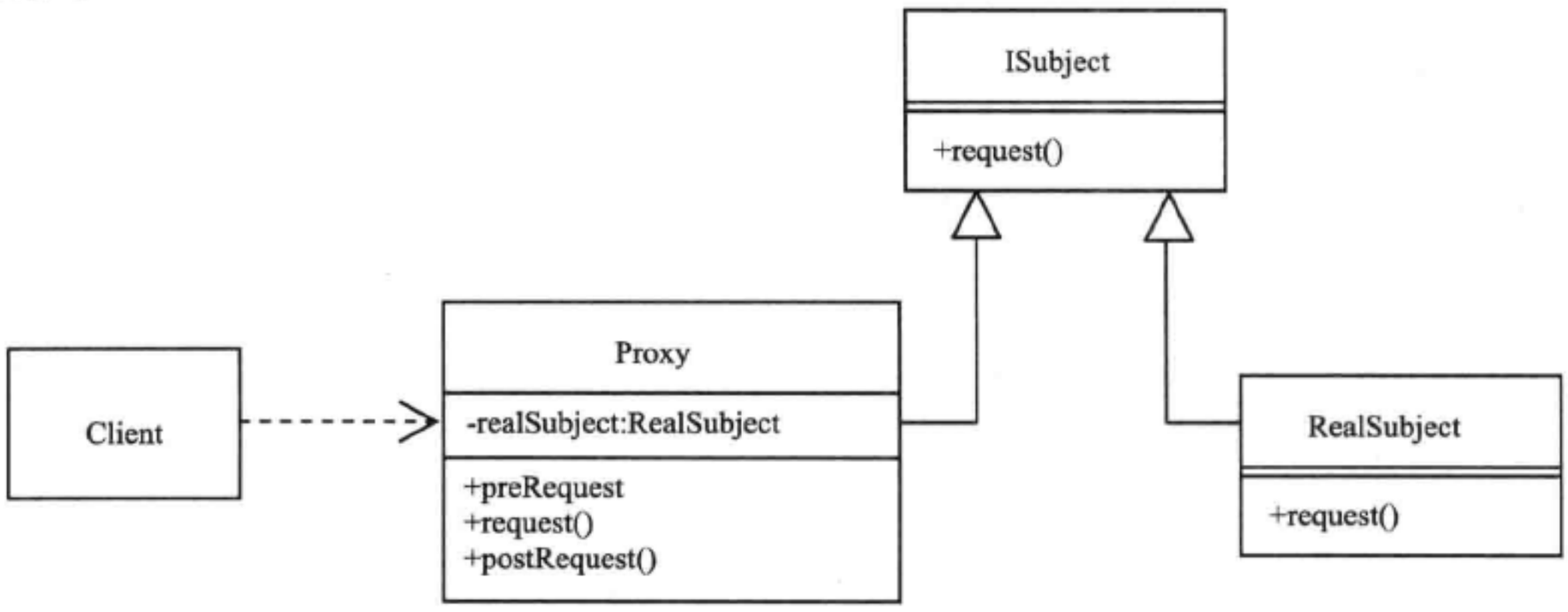


图 7-1 代理模式抽象 UML 类图

代理模式包含如下角色。

- ISubject：抽象主题角色，是一个接口。该接口是对象和它的代理共用的接口。

- RealSubject：真实主题角色，是实现抽象主题接口的类。
- Proxy：代理角色，内部含有对真实对象 RealSubject 的引用，从而可以操作真实对象。代理对象提供与真实对象相同的接口，以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

以买电视为例，其代码如下。

（1）定义抽象主题——买电视。

```
interface ITV{
    public void buyTV();
}
```

（2）定义实际主题——买电视过程。

```
class Buyer implements ITV{
    public void buyTV(){
        System.out.println("I have bought the TV by buyer proxy");
    }
}
```

真正的付费是由购买者完成的。

（3）定义代理。

```
class BuyerProxy implements ITV{
    private Buyer buyer;
    public BuyerProxy(Buyer buyer){
        this.buyer = buyer;
    }
    public void buyTV(){
        preProcess();
        buyer.buyTV();
        postProcess();
    }
    public void preProcess(){
        //询问客户需要电视类型、价位等信息
    }
    public void postProcess(){
        //负责把电视送到客户家
    }
}
```

电视代理商 BuyerProxy 与购买者 Buyer 都实现了相同的接口 ITV，是对 Buyer 对象的进一步封装。着重理解 buyTV()方法：首先代理商要通过 preProcess()询问客户买电视的类型、价位等信息，然后购买者通过 buyer.buyTV()自己付费完成电视购买,最后代理商通过 postProcess()协商具体的送货服务、产品三包等。

代理模式最突出的特点是：代理角色与实际主题角色有相同的父类接口。常用的代理方式有 4 类：虚拟代理、远程代理、计数代理、动态代理，下面一一加以说明。

7.3 虚 拟 代 理

虚拟代理的关键思想是：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。当用户请求一个“大”对象时，

虚拟代理在该对象真正被创建出来之前扮演着替身的角色；当该对象被创建出来之后，虚拟代理就将用户的请求直接委托给该对象。

【例 7-1】 高校本科生科研信息查询功能设计。

为了提高学生实践能力，高校教师每年都可以申请“开放实验室”项目，申报书包含的主要内容如表 7-1 所示。

表 7-1　　放实验室数据库表字段说明

序　号	字 段 名	关 键 字	说　明
1	account	√	账号
2	name		主持人姓名
3	project		项目名称
4	content		项目主要内容
5	plan		计划安排

可以看出，第 4、第 5 个字段均是大段的文字描述，如果直接列出申请的所有项目信息，一方面花费的时间会较长，另一方面界面也难设计，这是因为前 3 个字段长度较短，后 2 个字段长度较长。毫无疑问，这样的查询是不适合应用的。良好的查询策略应该分为二级查询。例如，第一级查询用表格形式显示“账号、姓名、项目名称”3 个字段。这相当于查询数据库表中的部分字段，而且字段短小，因此速度是较快的；第二级查询是当用鼠标选中表中某一个具体的项目时，再进行一次数据库查询，得到该项目的完整信息，在界面上显示第 4、第 5 个字段的内容。

初始界面如图 7-2（完成了第 1 级查询）所示。

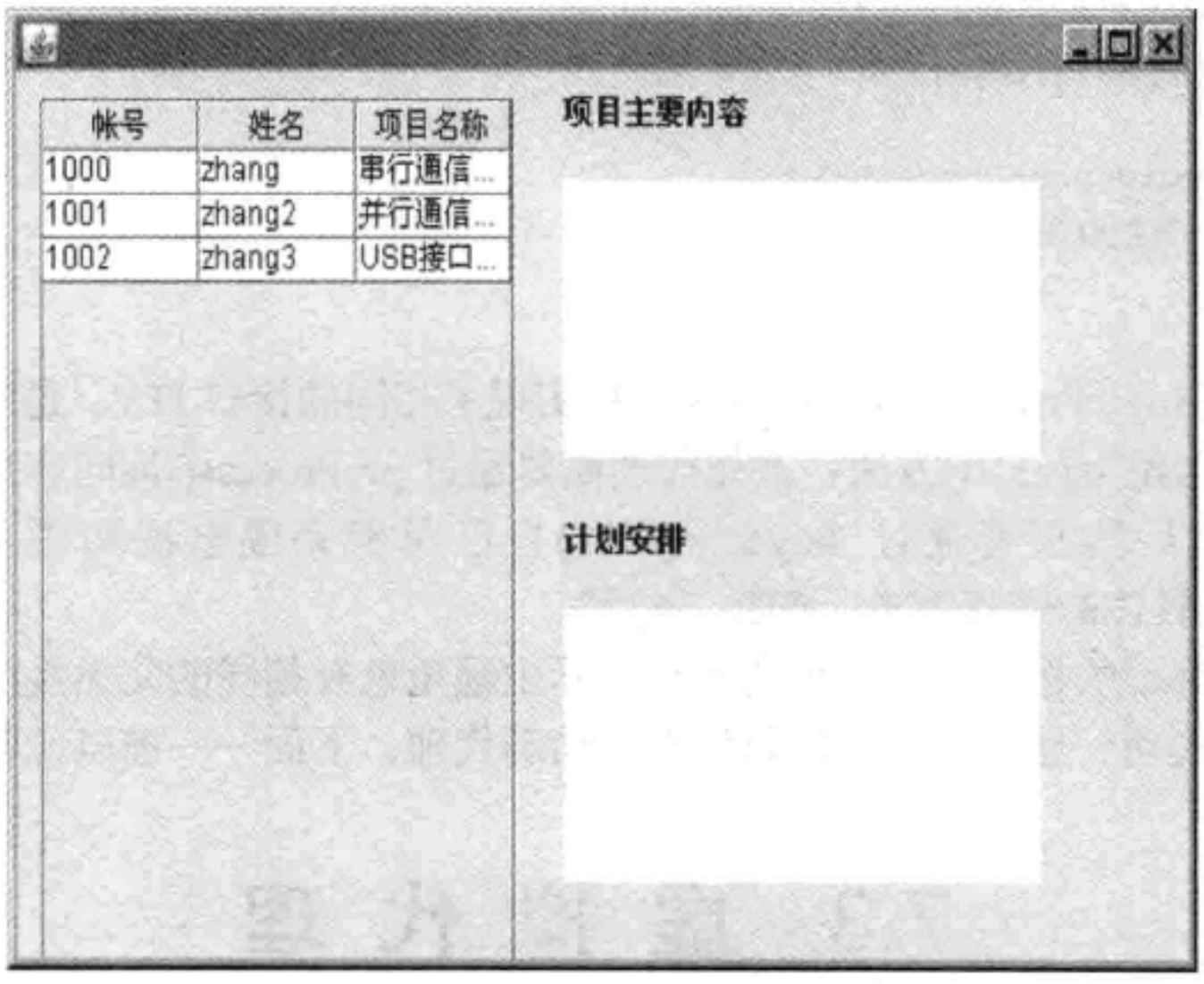


图 7-2　初始界面示意图

当用鼠标选中某一具体项目时，例如选中账号“1000”对应的项目时，则在界面右侧显示“项目主要内容”及“计划安排”内容，如图 7-3 所示。

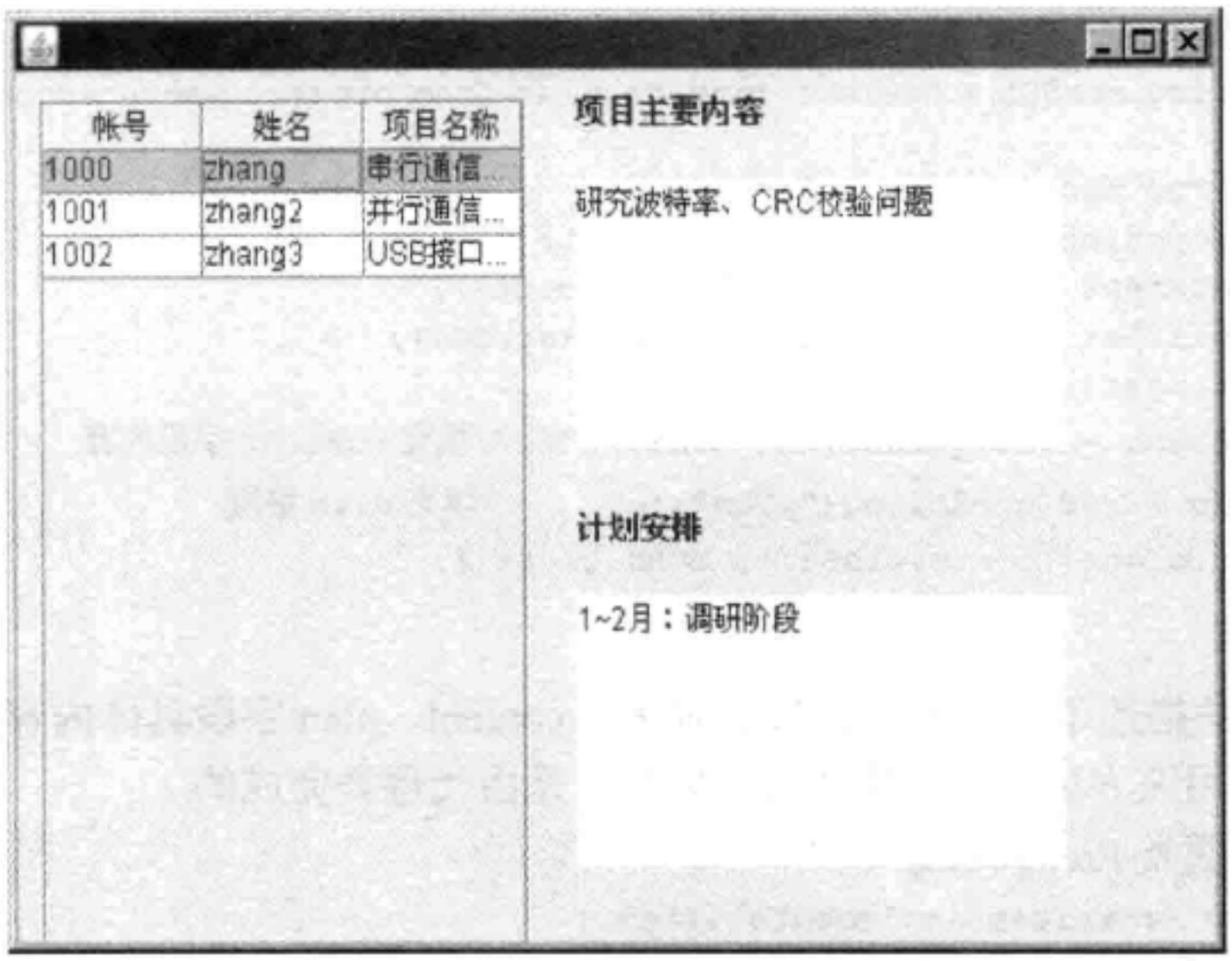


图 7-3 鼠标响应界面示意图

很明显，该示例可用代理模式完成，具体代码如下。

（1）定义抽象主题接口 IItem。

```
public interface IItem {
    String getAccount();  void setAccount(String s);
    String getName();void setName(String s);
    String getProject();  void setProject(String s);
    String getContent();
    String getPlan();
    void itemFill() throws Exception;
}
```

该接口表明主题是以一个项目的所有项信息为单位的。前三个字段 account、name、project 有 setter、getter 方法，表明这三个字段的信息是初始化时直接从数据库表得到的；而 content、plan 字段只有 getter 方法，表明这两个字段的信息仅当执行第二级查询时才填充完毕，由 itemFill()方法完成。

（2）具体实现主题类 RealItem。

```
public class RealItem implements IItem {
    private String account;
    private String name;
    private String project;
    private String content;
    private String plan;

    public String getAccount() {return account;}
    public void setAccount(String account) {this.account = account;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public String getProject() {return project;}
    public void setProject(String project) {this.project = project;}
    public String getContent() {return content;}
    public String getPlan() {return plan;}
    public void itemFill() throws Exception{     //填充本项目 content 及 plan 字段
```

```
//第 2 级查询 SQL 语句
String strSQL = "select content,plan from project where account='" +account+
                                                                    "'";
DbProc dbobj = new DbProc();
Connection conn = dbobj.connect();
Statement stm = conn.createStatement();
ResultSet rst = stm.executeQuery(strSQL);
rst.next();
content = rst.getString("content");//填充 content 字段内容
plan = rst.getString("plan");       //填充 plan 字段
rst.close(); stm.close(); conn.close();
    }
}
```

itemFill()方法描述了第二级查询时如何获得 content、plan 字段具体内容的过程，但是何时调用、怎样调用该方法在该类中并没有体现，是由代理类完成的。

（3）代理主题类 ProxyItem。

```
public class ProxyItem implements IItem {
    private RealItem item;
    boolean bFill;   //标识 content、plan 字段是否填充
    public ProxyItem(RealItem item){
        this.item = item;
    }
    public String getAccount() {
        return item.getAccount();
    }
    public void setAccount(String s) {
        item.setAccount(s);
    }
    public String getName() {
        return item.getName();
    }
    public void setName(String s) {
        item.setName(s);
    }
    public String getProject() {
        return item.getProject();
    }
    public void setProject(String s) {
        item.setProject(s);
    }
    public String getContent() {
        return item.getContent();
    }
    public String getPlan() {
        return item.getPlan();
    }
    public void itemFill() throws Exception {
        if(!bFill)
            item.itemFill();
        bFill = true;
    }
}
```

成员变量 bFill 是布尔变量，当执行 itemFill()时，由于初始 bFill 是 false，所以调用一次

具体主题类的 item.itemFill()方法，完成对 content、plan 字段的数据填充，最后把 bFill 置成 true。也就是说，无论调用代理对象 itemFill()多少次，由于 bFill 布尔变量的作用，只执行具体主题类对象的 itemFill()方法一次。

由于第一级是一个“大”范围的查询，因此一定是 ProxyItem 的集合。这就是下面描述的 ManageItems 类。

（4）代理项目集合类 ManageItems。

```
public class ManageItems {
    Vector<ProxyItem> v = new Vector();                          //代理项目集合
    public void firstSearch()throws Exception{
        String strSQL = "select account,name,project from project";//第1级查询SQL语句
        DbProc dbobj = new DbProc();
        Connection conn = dbobj.connect();
        Statement stm = conn.createStatement();
        ResultSet rst = stm.executeQuery(strSQL);                //获得第1级查询记录集合
        while(rst.next()){
            ProxyItem obj = new ProxyItem(new RealItem());
            obj.setAccount(rst.getString("account"));
            obj.setName(rst.getString("name"));
            obj.setProject(rst.getString("project"));
            v.add(obj);
        }
        rst.close(); stm.close(); conn.close();
    }
}
```

对于数据库应用程序来说，往往是对记录的集合进行操作的。因此若用到代理模式，则一般有一个代理基本类如 ProxyItem，代理集合管理类如 ManageItems。

（5）界面显示及消息映射类 UFrame。

```
public class UFrame extends JFrame implements MouseListener{
    ManageItems manage = new ManageItems();
    JTable table;
    JTextArea t = new JTextArea();
    JTextArea t2= new JTextArea();
    public void init()throws Exception{
        setLayout(null);
        manage.firstSearch(); //进行第1级查询

        String title[] = {"账号","姓名","项目名称"};
        String data[][] = null;
        Vector<ProxyItem> v = manage.v;
        data = new String[v.size()][title.length];
        for(int i=0; i<v.size(); i++){
            ProxyItem proxy = v.get(i);
            data[i][0] = proxy.getAccount();
            data[i][1] = proxy.getName();
            data[i][2] = proxy.getProject();
        }

        table = new JTable(data, title);
        JScrollPane pane = new JScrollPane(table);
        pane.setBounds(10, 10, 200, 340);
```

```
JLabel label = new JLabel("项目主要内容");
JLabel label2= new JLabel("计划安排");
label.setBounds(230,5,100,20); t.setBounds(230, 40, 200, 100);
label2.setBounds(230,160,100,20); t2.setBounds(230, 195, 200, 100);

add(pane);
add(label); add(t);
add(label2);add(t2);

table.addMouseListener(this);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(500,350);
    setVisible(true);
}
public void mouseClicked(MouseEvent event) {//进行第 2 级查询
    try{
        int n = table.getSelectedRow();    //Jtable 中方法
        if(n>=0){
            ProxyItem item = manage.v.get(n);
            item.itemFill();
            t.setText(item.getContent());
            t2.setText(item.getPlan());
        }
    } catch(Exception e){}
}
    public static void main(String[] args)throws Exception {
        new UFrame().init();
    }
}
```

7.4 远 程 代 理

远程代理的含义是：为一个位于不同的地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以在同一台主机中，也可在另一台主机中。也就是说，远程对象驻留于服务器上，客户机请求调用远程对象调用相应方法，执行完毕后，结果由服务器返回给客户端。

远程代理的基本框图如图 7-4 所示。

框图中虚线部分即远程代理。虚拟代理一般有一个代理，而远程代理包括两个代理：客户端通信代理与服务器端通信代理，因此编程中必须考虑这两部分因素，但是实际上 JDK 已经提供了远程代理的通信框架，如 RMI(Remote Method Invocation)远程方法调用、CORBA(Common Object Broker Architecture)公用对象请求代理体系结构技术等。这意味着编程时无须考虑远程代理的编码了，只编制所需的普通代码即可，甚至比虚拟代理编码都要简单许多。可能有读者疑问：远程代理部分负责信息的编码、传递、解码等功能，应该是非常复杂的，怎么可能被屏蔽掉，在编程时不考虑呢？其实，联系生活实际，就能有很好的感性认识。例如，你从大连邮寄包裹到北京，只需到大连邮局申请并登记一下就可以了，无须考虑北京到大连是如何邮寄的，那是“大连邮局代理”与“北京邮局代理”负责的事。因此，

远程代理也一定能做到像生活示例一样，客户端申请远程调用，而无须考虑如何调用的过程。至于更细致的理论说明，后文还有论述。

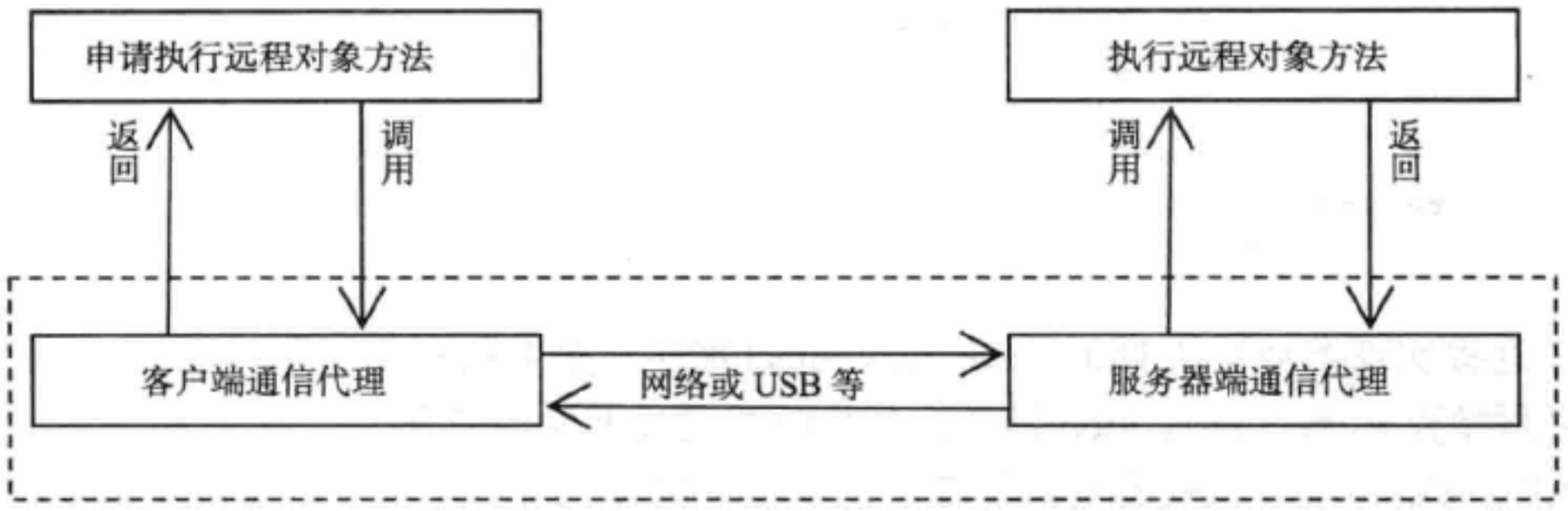


图 7-4 远程代理基本框图

7.4.1 RMI 通信

考虑一个简单的示例，客户端输入字符串数学表达式，仅含+、-运算。服务器端计算该表达式值，从前向后依次运行。

1. 创建服务器工程 RmiServer

（1）定义抽象主题远程接口 ICalc。

```
public interface ICalc extends Remote {
    float calc(String s)throws RemoteException;
}
```

RMI 远程接口必须从 Remote 接口派生，而且定义的接口方法中必须抛弃 RemoteException 异常。

（2）定义具体远程主题实现 ServerCalc。

```
public class ServerCalc extends UnicastRemoteObject implements ICalc {
    public ServerCalc()throws RemoteException{
        super();
    }
    public float calc(String s) throws RemoteException {
        s += "+0";
        float result = 0;           //最终结果变量
        float value = 0;            //拆分字符串对应的浮点变量
        char opcur = '+';           //当前操作符
        char opnext;
        int start = 0;              //字符串遍历起始位置
        if(s.charAt(0)=='-'){       //若是负数开始，则
            opcur = '-';            //修改当前操作符
            start = 1;              //修改字符串遍历起始位置
        }
        //遍历字符串
        for(int i=start; i<s.length(); i++){
            if(s.charAt(i)=='+' || s.charAt(i)=='-'){
                opnext = s.charAt(i);
                value = Float.parseFloat(s.substring(start, i));//按操作符拆分字符串
                                                                //对应数值
```

```
                switch(opcur){
                case '+': result += value; break;
                case '-': result -= value; break;
                }
                start = i+1; opcur = opnext;
                i = start;
            }
        }
        return result;
    }
}
```

RMI 远程实现类必须从 UnicastRemoteObject 派生，对本题而言还要实现 ICalc 接口；类中方法必须抛弃 RemoteException 异常，而且必须定义无参构造方法。

calc()完成字符串表达式的具体计算，主要理解第 1 行语句“s+= ‘+0’”,假设 s 初始为“1+2”，则变为“1+2+0”，主要是为了简化方法中 for 循环的边界条件处理，这也算是一个小的技巧，希望读者加以体会。

（3）远程具体实现对象注册类 RmiServer。

```
public class RmiServer {
    public static void main(String[] args) {
        if(System.getSecurityManager()==null)
            System.setSecurityManager(new RMISecurityManager());
        try{
            ServerCalc obj = new ServerCalc();
            Naming.rebind("calc", obj);
            System.out.println("server bind success!!!!!");
        }
        catch(Exception e){e.printStackTrace(); }
    }
}
```

main()中只是完成了远程对象的注册过程，是由 Naming 类中的静态方法 rebind()完成的。这表明 ServerCalc 是一个远程对象类，obj 是一个远程对象，注册名是字符串“obj”。这些只是完成 RMI 远程通信必要的准备工作，而且 main()方法执行后很快就结束了，并不是读者所想到的无限循环结构。其实，rebind()方法是完成向 RMI 应用服务器的注册。真正的 RMI 应用服务器执行后一定是无限循环结构，也一定是在该服务器运行后，才能完成远程对象的真实注册。JDK 中 RMI 应用服务器指 rmiregistry.exe，它位于 JDK 安装目录下面的 bin 子目录里，其默认端口号是 1099。

另外，RMI 通信必须进行安全管理，防止任意用户随意进行远程调用，基本方法是采用安全策略文件。它是一个文本文件，定义了远程访问的 IP 相关信息。本示例中服务器端用到的安全策略文件 server.policy 描述如表 7-2 所示。

表 7-2　　服务器端安全策略文件 server.policy

内　　容	说　　明

grant{
permission java.net.SocketPermission
"localhost:1099","connect,accept,listen,resolve";
permission java.net.SocketPermission
"210.47.223.8:80","connect,accept,listen,resolve";
};
	• 本机 RMI 应用服务器允许本机 1099 端口访问。
	• 本机 RMI 应用服务器允许指定 IP 端口访问

（4）执行服务器端注册功能。

为了方便，在 cmd 窗口下运行。前提条件是 path、classpath 环境变量都已设置完毕。

第 1 步，新开一个 cmd 窗口，运行: start rmiregistry，则出现图 7-5 所示界面。

图 7-5 RMI 注册服务器启动界面

第 2 步，新开一个 cmd 窗口，输入下述命令行（必须有安全策略文件选项）即可。

```
java -Djava.sevurity.policy=server.policy RmiServer
```

2. 创建客户端工程 RmiClient

（1）定义抽象主题远程接口 ICalc。

```
public interface ICalc extends Remote {
    float calc(String s)throws RemoteException;
}
```

必须与服务器端定义的 ICalc 接口一致，特别注意包目录必须完全相同。

（2）远程调用测试类。

```
public class RmiClient {
    public static void main(String[] args) {
        if(System.getSecurityManager()==null)
            System.setSecurityManager(new RMISecurityManager());
        try{
            ICalc obj = (ICalc)Naming.lookup("rmi://localhost:1099/calc");//查询远
                                                                          //程对象
            System.out.println(obj.calc("1+2+3")); //输出远程对象结果
        }
        catch(Exception e){e.printStackTrace();}
    }
}
```

客户端通过 Naming 类中的静态方法 lookup()查询远程对象，方法字符串参数定义为: “rmi://IP:Port/” + 远程对象注册字符串名称。

当然，RMI 客户端也采用安全策略文件，与表 7-2 中内容相仿，假设策略文件名为 client.policy，则客户端执行命令行如下。

```
Java -Djava.security.policy=client.policy RmiClient
```

7.4.2 RMI 代理模拟

7.4.1 小节中编制的 RMI 程序均没有显示的代理程序存在，因为 JDK 把代理程序给屏蔽掉了。为了更好地理解远程代理程序的作用，仍以上节远程运算为例，编写一个简单的 RMI 代理模拟程序。

1. 创建服务器工程 RmiServerSimu

（1）定义抽象主题远程接口 ICalc。

```
public interface ICalc{
```

```
    float calc(String s)throws Exception;
}
```

ICalc 无须从 Remote 派生。

（2）定义具体远程主题实现 ServerCalc。

```
public class ServerCalc implements ICalc {
    public float calc(String s) throws Exception {
        //代码同 7.4.1 节
        }
        return result;
    }
}
```

ServerCalc 类无须从 UnicastRemoteObject 派生。

（3）定义服务器端远程代理类 ServerProxy。

```
public class ServerProxy extends Thread {
    ServerCalc obj;
    public ServerProxy(ServerCalc obj){
        this.obj = obj;
    }
    public void run(){
        try{
            ServerSocket s = new ServerSocket(4000);
            Socket socket = s.accept();
            while(socket != null){
                ObjectInputStream in =
                                    new ObjectInputStream(socket. getInputStream());
                ObjectOutputStream out =
                                    new ObjectOutputStream(socket. getOutputStream());
                String method = (String)in.readObject();    //读取方法字符串
                if(method.equals("calc")){
                    String para = (String)in.readObject(); //读取方法参数
                    float f = obj.calc(para);              //计算出表达式值
                    out.writeObject(new Float(f));         //返回给客户端
                    out.flush();
                }
            }
        }
        catch(Exception e){e.printStackTrace();}
    }
}
```

一般的应用服务器一定是支持多线程的，故定义 ServerProxy 从 Thread 派生。RMI 服务器端程序是通过网络通信的，所以用到了 Socket 接口。run()中封装了远程代理的功能，即主要是通过 ObjectInputStream、ObjectOutputStream 对象 in、out 读写网络。当客户端申请远程执行 calc(String expression)方法时，in 首先读取方法名 method；若 method 为字符串 “calc”，in 继续读取网络，获得表达式字符串 expression。然后调用 ServerCalc 类中的 calc()方法，计算出表达式 expression 的值 value，最后通过 out 对象传送回客户端。

（4）一个简单的测试类 RmiServerSimu。

```
public class Test {
    public static void main(String[] args) {
        ServerCalc obj = new ServerCalc();          //定义具体实现类
```

```
        ServerProxy spobj = new ServerProxy(obj);    //定义服务器远程代理类
        spobj.start();                               //启动线程
    }
}
```

2. 创建客户端工程 RmiClientSimu

（1）定义抽象主题远程接口 ICalc。

```
public interface ICalc{
    float calc(String s)throws Exception;
}
```

（2）定义服务器端远程代理类 ClientProxy。

```
public class ClientProxy implements ICalc {
    Socket socket ;
    public ClientProxy()throws Exception{
        socket = new Socket("localhost",4000);
    }
    public float calc(String expression) throws Exception {
        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
        out.writeObject("calc");
        out.writeObject(expression);
        Float value = (Float)in.readObject();
        return value.floatValue();
    }
}
```

ClientProxy 是从远程接口 ICalc 派生的（注意：ServerProxy 不从 ICalc 派生）。RMI 客户端程序是通过网络通信的，所以用到了 Socket 接口。calc()中体现了远程代理的功能，即主要是通过 ObjectInputStream、ObjectOutputStream 对象 in、out 读写网络。首先通过 out 对象向服务器端写申请执行的函数名“calc”，然后写表达式 expression，最后通过 in 对象从服务器端取得计算结果，并返回给调用端。

（3）一个简单的测试类 RmiClientSimu。

```
public class ClientTest {
    public static void main(String[] args) throws Exception{
        ICalc obj = new ClientProxy();          //客户端代理对象初始化
        float value = obj.calc("23+2+3");       //通过远程代理计算表达式的值
        System.out.println("value=" +value);
    }
}
```

我们发现，ClientProxy、ServerProxy 类中的功能是相似的。理解了这两个类的含义，再编制其他的远程方法，代码也是大同小异。因此，JDK 专家人员进行进一步抽象，编制了系统的 RMI 通信代码。

7.5 计 数 代 理

当客户程序需要在调用服务提供者对象的方法之前或之后执行日志或计数的额外功能时，就可以使用计数代理模式。计数代理模式并不是把这些额外操作的代码直接添加到源服

务中，而是把它们封装成一个单独的对象，这就是计数代理。

考虑这样一个应用，用计数代理统计图书馆中每天借阅书籍的具体次数，代码如下。

1. 定义书籍基本类 Book

```
class Book{
    private String NO;     //书号，假设仅有书号、书名两个属性
    private String name;  //书名
    public Book(String NO, String name){
        this.NO = NO;
        this.name = name;
    }
    public String getNO() {
        return NO;
    }
    public void setNO(String nO) {
        NO = nO;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

2. 定义抽象主题接口 IBorrow

```
interface IBorrow{
    boolean borrow();   //借阅过程
}
```

3. 定义借阅实现类 Borrow

```
class Borrow implements IBorrow{
    private Book book;
    public void setBook(Book book) {
        this.book = book;
    }
    public Book getBook(){
        return book;
    }
    public boolean borrow(){
        //保存信息到数据库等功能，代码略
        return true;
    }
}
```

4. 定义借阅代理类 BorrowProxy

```
class BorrowProxy implements IBorrow{
    private Borrow obj;
    private Map<String, Integer> map;
    public BorrowProxy(Borrow obj){
        this.obj = obj;
        map = new HashMap();
    }
    public boolean borrow(){
```

```
        if(!obj.borrow())      //借阅失败则
            return false;      //返回
        Book book = obj.getBook();
        Integer i = map.get(book.getNO());
        i=(i==null)?1:i+1;
        map.put(book.getNO(), i);//保存“书号-次数”键-值对
        return true;
    }
    public void log()throws Exception{
        Set<String> set = map.keySet();
        String key = "";
        String result = "";
        Iterator<String> it = set.iterator();
        while(it.hasNext()){
            key = it.next();
            result += key +"\t"+ map.get(key) +"\r\n";
        }
        Calendar c = Calendar.getInstance();
        RandomAccessFile fa = new RandomAccessFile("d:/log.txt","rw");
        fa.seek(fa.length());
        fa.writeBytes(+c.get(Calendar.YEAR)+"-"+(c.get(Calendar.MONTH)+1)+"-"+
c.get(Calendar.DAY_OF_MONTH)+"\r\n"); //记录日志时间
        fa.writeBytes(result);     //记录每本书对应借阅次数
        fa.close();
    }
}
```

该代理类定义了 Map 成员变量 map，键是书号，每天借阅次数是值。日志文件中首先保存当前时间，然后保存“书号\t 借阅次数信息”，一行一条记录。

5. 一个简单测试类

```
public class Test {
    public static void main(String[] args)throws Exception {
        Borrow br = new Borrow();
        BorrowProxy bp = new BorrowProxy(br);

        Book book = new Book("1000", "计算机应用");
        br.setBook(book);
        bp.borrow(); //借阅一本书

        book = new Book("1001", "计算机应用 2");
        br.setBook(book);
        bp.borrow(); //借阅另一本书

        bp.log();     //调用记录日志功能
    }
}
```

可以看出，本示例并没有实现每天都记录日志一次，因为从设计思想角度来说，时间控制应该在主框架实现，到约定时间，调用代理类的 log()方法即可。如果硬要在代理类添加时间控制，也是可行的，控制好 JDK 中的 Timer 定时器就可以了，在此就不作论述了。

7.6 动 态 代 理

7.6.1 动态代理的成因

一般来说，对代理模式而言，一个主题类与一个代理类一一对应，这也是静态代理模式的特点。其程序模型如图 7-6 所示（假设有 *n* 个主题类）。

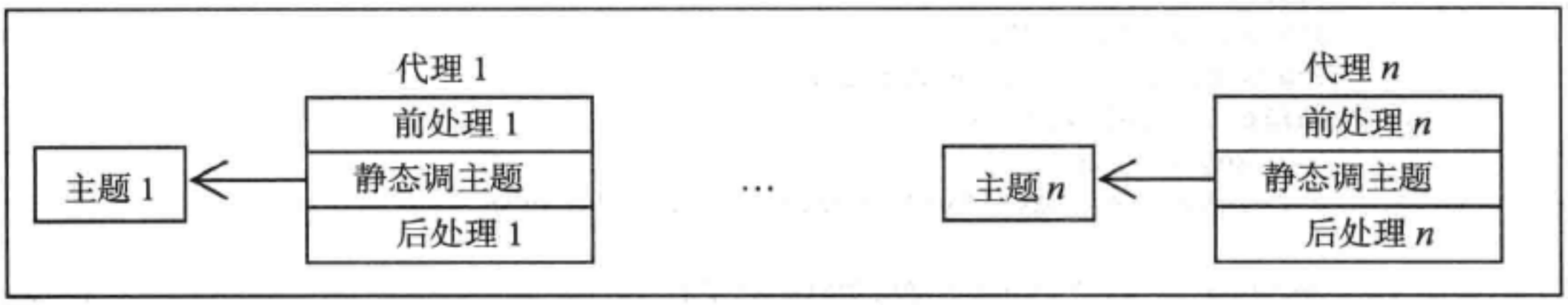


图 7-6 静态代理模式简图

但是，也常存在这样的情况，有 *n* 个主题类，但是代理类中的“前处理、后处理”都是一样的，仅调用主题不同，需要编制图 7-7 所示的程序框架。

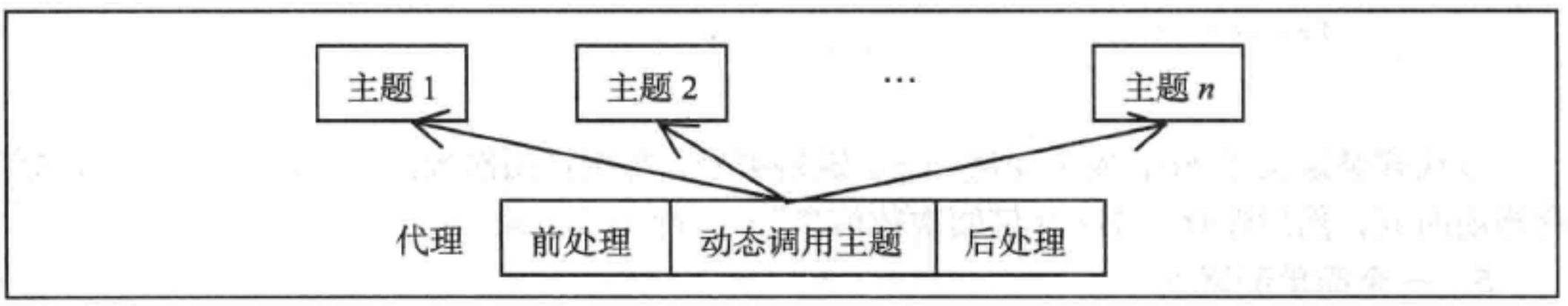


图 7-7 动态代理模式简图

也就是说，多个主题类对应一个代理类，共享“前处理、后处理”功能，动态调用所需主题，大大减小了程序规模，这就是动态代理模式的特点。

实现动态代理的关键技术是反射。反射知识在第 2 章已有详细介绍，下面主要讲述反射在代理模式中的具体应用方法。

7.6.2 自定义动态代理

重新考虑 7.4.2“RMI 代理模拟”一节内容，该功能完全可由动态代理模式实现，其具体过程如下。

1. 创建服务器工程 URmiServer

（1）定义抽象主题远程接口 ICalc。

```
public interface ICalc{
    float calc(String s)throws Exception;
}
```

（2）定义具体远程主题实现 ServerCalc。

```
public class ServerCalc implements ICalc {
    //同 7.4.2 小节
```

```
}
```

（3）定义服务器端远程代理类 ServerProxy。

```
public class ServerProxy{
    public static Map<String, Object>map = new HashMap();
    public void registry(String key, Object value){   //远程对象注册功能
        map.put(key, value);                           //注册到 HashMap 映射中
    }
    Public void process(int socketNO)throws Exception{
        ServerSocket s = new ServerSocket(socketNO);
        while(true){
            Socket socket = s.accept();
            if(socket != null){
                MySocket ms = new MySocket(socket);
                ms.start();
            }
        }
    }
}
```

该类是远程代理通信功能的核心类。通过 registry()方法，把主题对象注册到成员变量 map 映射中，方便将来获得该对象，并利用反射机制执行该对象中的方法。process()方法表明 ServerProxy 是一个多线程类，常规线程负责侦听客户端的连接；若有连接，则获得该连接；并创建 MySocket 线程对象，然后启动该线程。

（4）MySocket 类。

```
public class MySocket extends Thread {
Socket socket;
public MySocket(Socket socket){
    this.socket = socket;
}
public Object invoke(String registname, String methodname, Object para[])
                            throws Exception{
    Object obj = ServerProxy.map.get(registname); //获得注册对象
    //形成函数参数序列
    Class classType = Class.forName(obj.getClass().getName());
    Class c[] = new Class[para.length];
    for(int i=0; i<para.length; i++){
        c[i] = para[i].getClass();
    }
    //利用反射机制调用主题对象方法
    Method mt = classType.getMethod(methodname, c);
    return mt.invoke(obj, para);
}
public void run() {
    while(true){
        try{
            InputStream ins = socket.getInputStream();
            if(ins==null || ins.available()==0)
                continue;
            //前处理
            ObjectInputStream in = new ObjectInputStream(ins);
```

```
ObjectOutputStream out =
            new ObjectOutputStream(socket.getOutputStream());
String registname = (String)in.readObject();  //获得远程对象注册
                                              //名称
String methodname = (String)in.readObject();  //获得远程调用方法
                                              //名称
Object[] para = (Object[])in.readObject();    //获得远程对象方法
                                              //参数数组

//动态调用主题对象
                Object result = invoke(registname,methodname,para);
                //后处理
                out.writeObject(result);                  //将结果写回客户端
                out.flush();
            }
            catch(Exception e){e.printStackTrace();}
        }
    }
}
```

线程运行 run()方法中包含了远程代理服务器端的主要功能："前处理"主要是三次读网络，第 1 次获取远程对象注册名称 registname，第 2 次获取远程调用方法名称 methodname，第 3 次获得远程方法实参数组 para；根据获得的三个值，利用反射机制完成"动态主题调用"，主要体现在 invoke()方法中；"后处理"负责把结果写回客户端。

对某功能来说，若它能用动态代理完成，则"前处理、后处理"功能的划分是最为关键的。希望读者借助本例仔细体会。

（5）一个简单的测试类。

```
public class URmiServer {
    public static void main(String[] args) throws Exception{
        ServerCalc obj = new ServerCalc();          //定义实现类
        ServerProxy spobj = new ServerProxy();      //定义代理类
        spobj.registry("calc", obj);                //注册实现类
        spobj.process();                            //进行处理过程循环
    }
}
```

可知，具体主题 ServerCalc 对象 spobj 的注册名称为"calc"，表明：若客户端相应传过来的字符串为"calc"，则表示要调用 spobj 中的方法。

2. 创建客户端工程 URmiClient

（1）定义抽象主题远程接口 ICalc。

```
public interface ICalc{
    float calc(String s)throws Exception;
}
```

（2）定义客户端计算代理 CalcProxy。

```
public class CalcProxy implements ICalc{
    ClientComm comm;
    public CalcProxy(String IP, int socketNO)throws Exception{
        comm = new ClientComm(IP,socketNO);
    }
```

```
    public float calc(String s) throws Exception {
        Float result = (Float)comm.invoke("calc", "calc", new Object[]{s});
        return result.floatValue();
    }
}
```

该类从接口 ICalc 派生，但 calc()方法中并没有真正实现求表达式的具体过程，只是把相关参数负责向远程服务器端传送，是由 ClientComm 类对象具体完成的。也就是说，发送任意远程方法所需相关参数都由该类完成，是共享的。其具体代码如下。

```
public class ClientComm {
    Socket socket;
    public ClientComm(String IP,int socketNO)throws Exception{
        socket= new Socket(IP, socketNO);
    }
    Object invoke(String registname, String methodname, Object[] para) throws
                        Exception {
        //前处理
        ObjectOutputStream out =
                            new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
        out.writeObject(registname);
        out.writeObject(methodname);
        out.writeObject(para);
        return in.readObject(); //后处理
    }
}
```

invoke()方法参数有 3 个，第 1 个是远程对象注册名称，第 2 个是远程方法名称，第 3 个是远程方法执行所需参数值序列，因此向网络写 3 次，然后读网络，等待返回结果即可。

（3）一个简单测试类。

```
public class URmiClient {
    public static void main(String[] args) throws Exception{
        ICalc obj = new ClientProxy(4000);
        System.out.println(obj.calc("1+5+10"));
        System.out.println(obj.calc("1+5+20"));
    }
}
```

通过本示例，读者可对动态代理有一个初步的理解，也可进一步懂得 JDK 能对 RMI 采用动态代理封装的原因。

当然，本示例还有一些缺陷，如客户端执行完毕后，没有通知服务器端断开 socket 通信；服务器端多线程同步问题、结束线程问题等还需要进一步完善。

7.6.3 JDK 动态代理

动态代理工具是 java.lang.reflect 包的一部分，在 JDK 1.3 版本中添加到 JDK，它允许程序创建代理对象。它能实现一个或多个已知接口，并用反射机制动态执行相应主题对象。

在 1.5 以前的 JDK 中，RMI 远程代理相当于静态代理，相关代码是在编译时由 RMI 编译器 rmic 生成的。对于每个远程接口，都会生成一个 stub（代理）类，它代表远程对象，还生成一个 skeleton 对象，叫作骨架类。它在远程 JVM 中做与 stub 相反的工作——解析传送

参数并调用实际的对象。在 1.5（包括 1.5）之后的 JDK 中，RMI 通信采用了动态代理，无须生成 stub 及 skeleton 对象，结果 RMI 编程就像本地编程一样，不必编制代理类，RMI 编程获得了极大的简化。

由于 RMI 通信已经有成熟的动态代理，因此从应用角度来说，动态代理更适于在虚拟代理、计数代理等中获得应用。

考虑这样一个应用：有两种渠道接收信件，一种是通过 Email 方式，另一种是通过传统的邮寄方式。对来的信件都要进行登记，现在要求利用代理模式增加功能，对不同方式来的信件进行记数统计。使用 JDK 动态代理仿真此功能的代码如下。

1. 定义抽象主题 IRegist

```
interface IRegist{
    void regist(String msg);   //对来的信件进行登记
}
```

2. 定义 2 个具体主题

```
class fromEmail implements IRegist{  //Email 信件登记类
    public void regist(String msg){
        System.out.println("from Email");
    }
}

class fromPost implements IRegist{ //传统邮寄信件登记类
    public void regist(String msg){
        System.out.println("from post");
    }
}
```

3. 定义动态代理相关类及接口

上文已经定义了两个具体的主题类：fromEmail，fromPost。如果是静态代理，那么一定定义两个代理类。由于功能是相同的，如果采用 JDK 动态代理，则只需要一个代理类，但必须严格按规范编程，具体步骤如下。

（1）定义计数实现类 CountInvoke。

```
class CountInvoke implements InvocationHandler{
    private int count = 0;          //计数变量
    private Object obj;             //具体主题对象
    public CountInvoke(Object obj){
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        count ++;
        method.invoke(obj, args);     //对主题对象 obj 应用反射技术调用相应方法
        return null;
    }
    int getCount(){                   //返回计数值
        return count;
    }
}
```

需要注意，该类并不是动态代理类，它是动态代理类所调用的接口实现类，因此接口不能随意，必须由系统指定，此接口名称为 InvocationHandler。

成员变量 obj 代表具体主题对象，它是 Object 类型，可泛指任意主题对象。因此，从广义来说，CountInvoke 可推广到对任意主题对象计数，前提条件是这些主题的“前处理、后处理”功能是相同的。

invoke()是接口 InvocationHandler 定义的方法，必须实现。该方法完成了代理所需要的功能，包括前处理、利用反射技术调用主题方法、后处理等。一般来说，实现动态代理，主要是完成接口方法 invoke()的编制。

（2）创建代理类 GenericProxy。

```
class GenericProxy{
    public static Object createProxy(Object obj, InvocationHandler invokeObj){
        Object proxy = Proxy.newProxyInstance(obj.getClass().getClassLoader(),
 obj.getClass().getInterfaces(), invokeObj);
        return proxy;
    }
}
```

主要是通过 createProxy()产生代理对象，其方法参数有两个：第 1 个参数 obj 表示具体主题对象，第 2 个参数 invokeObj 表示代理调用的接口实现类对象。说得更通俗一些就是，为主题对象 obj 创建代理对象，并与调用接口 invokeObj 对象建立关联，以便将来代理对象能调用接口方法。

由 createProxy()可知，真正产生动态代理对象的是 JDK 中 Proxy 类的静态方法 newProxyInstance()。由于其第 2 个参数要求主题对象的父类必须是接口，所以目前 JDK 动态代理前提条件是抽象主题必须是接口。

（3）一个简单测试类 Test。

```
public class Test {
    public static void main(String[] args) {
        IRegist email = new fromEmail();          //产生邮件主题对象
        IRegist post = new fromPost();            //产生邮寄主题对象
        CountInvoke emailinvoke = new CountInvoke(email);    //邮件代理对象调用的接口
                                                             //对象
        CountInvoke postinvoke = new CountInvoke(post);      //邮寄代理对象调用的接口
                                                             //对象

        IRegist emailproxy = (IRegist)GenericProxy.createProxy(email, emailinvoke);
                        //email 代理对象
        IRegist postproxy = (IRegist)GenericProxy.createProxy(post, postinvoke);;
                        //邮寄代理对象

        emailproxy.regist("email1");   //登记从 email 来的信件
        postproxy.regist("post1");     //登记从邮寄来的信件
        System.out.println(emailinvoke.getCount()); //显示 email 登记的信件数目
        System.out.println(postinvoke.getCount());  //显示邮寄登记的信件数目
    }
}
```

第8章 状态模式

8.1 问题的提出

生活中有一类事物，有 n 种状态，每种状态下有不同的特征，在一定的条件下，状态间可以相互转化，如图 8-1 所示。

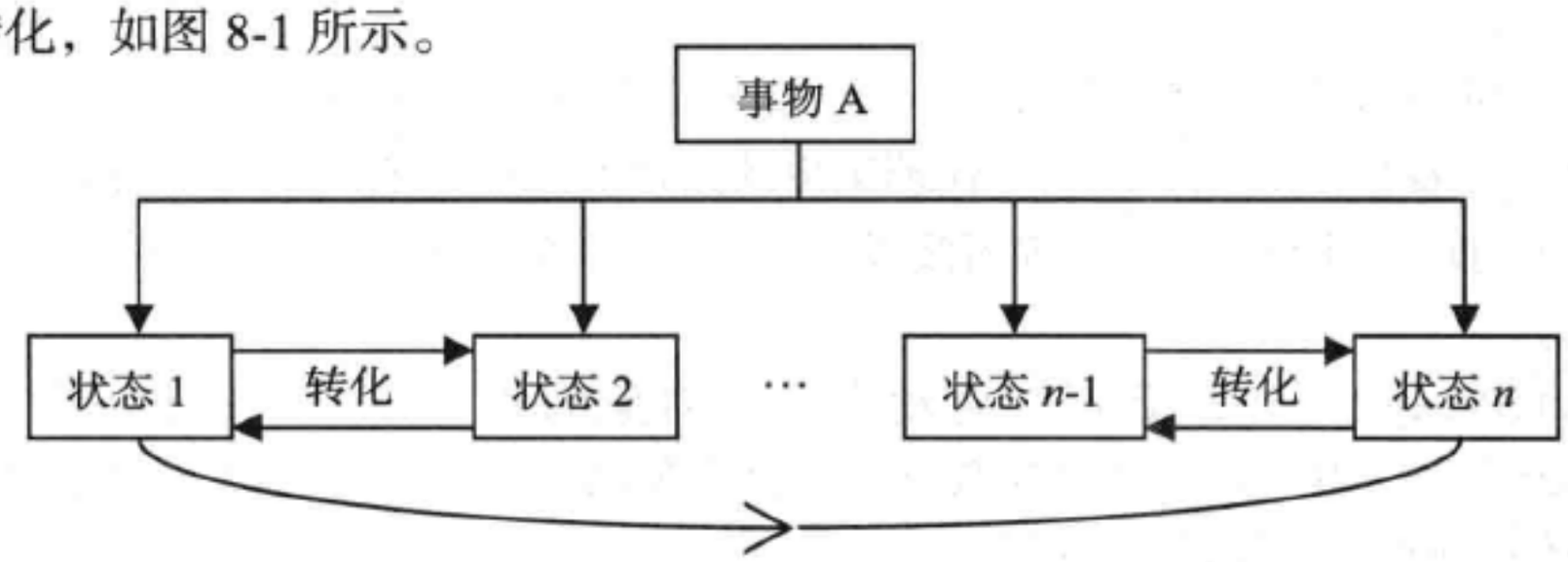


图 8-1 事物状态转化图

从图 8-1 分析可以看出，初始化时事物 A 位于状态 m（$m \leqslant n$），在满足一定条件的时候，状态间可相互转化。如状态 1 可以转化为状态 2，状态 2 也可以转化为状态 1，也可能状态 1 直接转化为状态 n。生活中类似的示例是很多的，如水有固、液、汽三态，当温度 $T \leqslant 0$℃时是固态，当 $0<T<100$℃时是液态，$T=100$℃时是气态；再如初始浏览某网站时处于游客状态，简单注册后升级为普通用户状态，也可能付费注册后，直接由游客状态变为 VIP 状态。

总之，研究各种状态以及状态间相互转化的实现方式是本章研究的关键问题，状态模式提出了一种较好的设计思路。

8.2 状 态 模 式

状态模式要求应反映图 8-1 的语义：事物有 n 个状态，且维护状态变化。从此句出发可以得出如下重要结论。

- 状态类有共同的父接口（或抽象类），n 个不同的状态实现类。
- 事物类中包含状态类父接口成员变量声明，用以反映语义“事物有 n 个状态”。
- 事物类中一定有方法选择分支，判断事物当前处于何种状态。

上述结论用专业语言描述，即状态模式必须完成如下内容的编制。

- State：状态接口，封装特定状态所对应的行为。
- ConcreteState：具体实现状态处理的类。
- Context：事物类，也称上下文类，通常用来定义多态状态接口，同时维护一个用来具体处理当前状态的示例对象。

状态模式的 UML 抽象类图如图 8-2 所示。

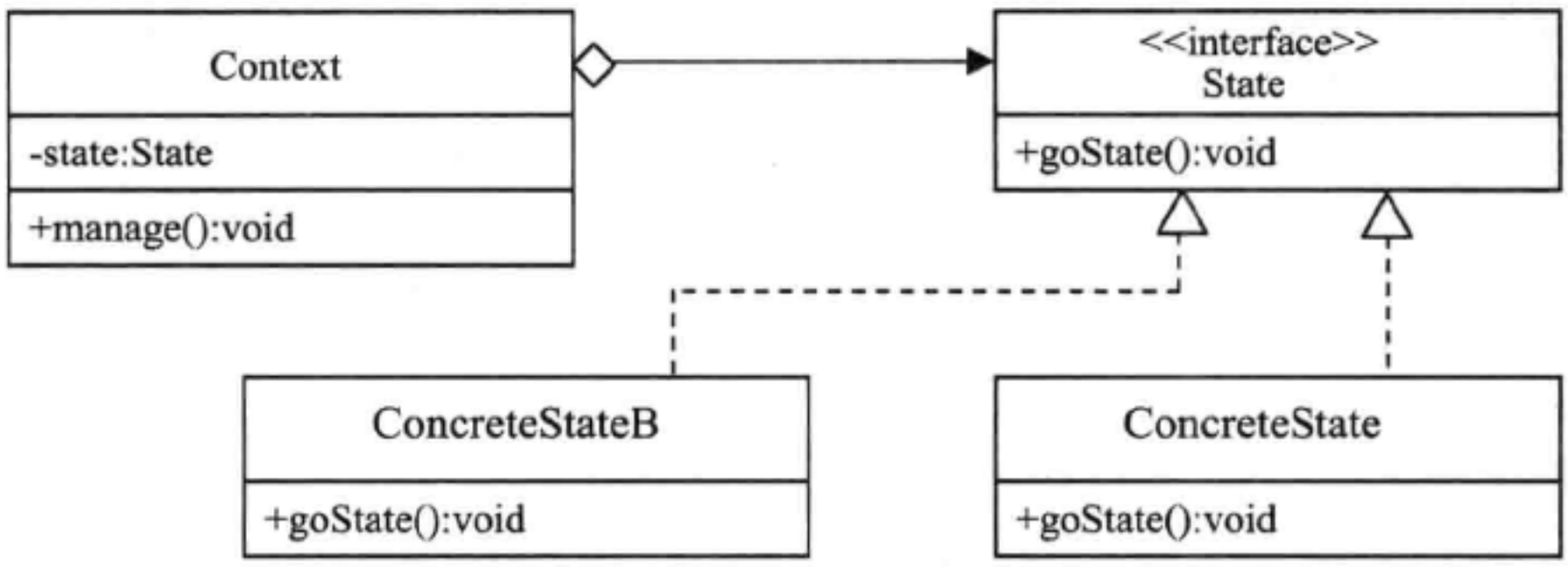


图 8-2 状态模式抽象 UML 类图

状态模式的具体抽象代码如下。

（1）定义状态抽象接口 IState。

```
interface IState{
    public void goState();
}
```

（2）定义状态实现类。

```
class ConcreteStateA implements IState{//定义状态 A 类
    public void goState(){
        System.out.println("This is ConcreteStateA");
    }
}
class ConcreteStateB implements IState{//定义状态 B 类
    public void goState(){
        System.out.println("This is ConcreteStateB");
    }
}
```

（3）定义状态上下文维护类 Context。

```
class Context{                    //上下文有 n 种状态
    private IState state;
    public void setState(IState state){
        this.state = state;
    }
    public void manage(){
        //此处代码根据条件选择某种状态
        state.goState();  //执行某种状态功能
    }
}
```

Context 类是实现状态模式的关键，本部分仅列出了状态模式的基本代码，希望读者有一个大致的了解。

8.3 深入理解状态模式

1. 利用上下文类控制状态

考虑手机应用。假设手机功能有存款、电话功能。有三种状态，正常、透支、停机三种状态，试用状态模式加以仿真描述。

（1）定义手机状态接口 ICellState。

```
interface ICellState{
    public float NORMAL_LIMIT = 0;          //正常状态
    public float STOP_LIMIT = -1;           //停机状态
    public float COST_MINUTE = 0.20f;       //话费标准
    public boolean phone(CellContext ct);
}
```

当手机余额> *NORMAL_LIMIT* 时，手机处于正常状态；当余额<*STOPL_LIMIT* 时，手机处于停机状态；当 *NORMAL_LIMIT*≥余额>*STOP_LIMIT* 时，手机处于透支状态。

按照需求分析，手机有存款、电话两个功能，那为什么接口仅定义 phone()方法呢？这是因为系统需要在数据库中记录每个人每次打电话的信息，信息存放的位置是不同的，正常打电话信息存入“正常表”中，透支打电话信息存入“透支表”中等，目的是将来方便做统计查询。试想，如果一个人总在透支状态下打电话，那么他的诚信度就会降低，将来信贷时就会遇到麻烦。经过以上分析，phone()功能是多态的。而存款功能属于基本信息，是非多态的，所以不必定义在接口中。

phone()方法参数类型是 CellContext，转化成普通的语义是：手机用户要打电话了。CellContext 是手机上下文类，封装了手机用户的相关信息。

（2）定义手机用户三种状态类。

```
//正常状态下打电话类
class NormalState implements ICellState{
    public boolean phone(CellContext ct) {
        System.out.println(ct.name +":手机处于正常状态");
        int minute = (int)(Math.random()*10+1); //随机产生打电话分钟数
        ct.cost(minute); //计算电话消费，cost()方法在 CellContext 类中实现
        //保存信息到数据库的正常表中
        return false;
    }
}
//透支状态下打电话类
class OverDrawState implements ICellState{
    public boolean phone(CellContext ct) {
        System.out.println(ct.name +":已处于欠费状态，请及时缴费");
        int minute = (int)(Math.random()*10+1);
        ct.cost(minute);
        //保存信息到数据库的透支表中
        return false;
    }
}
```

```
//停机类
class StopState implements ICellState{
    public boolean phone(CellContext ct) {
        System.out.println(ct.name +":已处于停机状态，请及时缴费");
        //保存信息到数据库
        return false;
    }
}
```

各种状态下打电话的仿真算法及流程是相似的:利用随机方法 random()产生打电话的分钟数，范围在[1，10]，然后计算手机余额，最后把信息保存到数据库的相应表中。数据库操作代码不在本部分讨论内容范围内，故省略。

（3）手机上下文状态类 CellContext。

```
class CellContext{
    String strPhone;  //电话号码
    String name;      //姓名
    float price;      //金额
    public CellContext(String strPhone, String name, float price){
        this.strPhone = strPhone; this.name = name; this.price = price;
    }
    public void save(float price){  //手机存钱
        this.price += price;
    }
    public void cost(int minute){  //手机打了 n 分钟，重新计算余额
        this.price -= ICellState.COST_MINUTE*minute;
    }
    public boolean call(){
        ICellState state = null;
        if(price > ICellState.NORMAL_LIMIT)
            state = new NormalState();
        else if(price < ICellState.STOP_LIMIT)
            state = new StopState();
        else
            state = new OverDrawState();
        state.phone(this); //调用 IState 接口中方法，该方法由不同子类实现
        return true;
    }
}
```

本类其实就是手机用户类，构造方法 CellContext()定义了用户的信息，包括手机号、所属人姓名、初始余额等。每次打电话时，手机状态完全是在该类中由 phone()方法根据余额确定的，与具体的状态类无关。和图 8-2 所示状态模式 UML 类图对应相关代码比较，本类中根本没有把状态接口作为成员变量进行封装。因此，当具体状态完全由上下文对象确定时，无须在上下文类中封装多态状态接口。请读者灵活掌握。

（4）一个简单的测试类。

```
public class Test {
    public static void main(String[] args) {
        CellContext c = new CellContext("1380908925","jin",1);   //新建手机用户，余额
                                                                 //1元
        c.call();c.call();                                       //打两次电话
```

```
            c.save(4);                                          //又存入 4 元钱
            c.call();c.call();c.call();c.call();                //又打 4 次电话
        }
    }
```

每次执行后结果可能不同，这是由于在具体状态类中采用随机方法生成打电话耗时时间的缘故。

（5）用集合管理类改善程序功能。

本例中 CellContext 类仅是手机用户的基础类，仅能产生单手机用户对象。而实际中必须对手机用户对象的集合加以维护，因此，编制了 CellContext 对象的集合管理类 Manage，如下所示。

```
class Manage{
    private Map<String,CellContext> map = new HashMap();
    public boolean regist(CellContext ct){  //注册新用户功能
        map.put(ct.strPhone, ct);
        return true;
    }
    public void save(String strPhoneNO, float money){//为源手机号 strPhoneNO 存入
                                                      //money 钱
        CellContext ct = map.get(strPhoneNO);
        ct.save(money);
    }
    public boolean phone(String strPhoneNO){//源手机号 strPhoneNO 申请打电话
        CellContext ct = map.get(strPhoneNO);
        ct.phone();
        return true;
    }
}
```

一个重新编制的测试类如下。

```
public class Test {
    public static void main(String[] args) {
        CellContext c = new CellContext("138409025","jin",1); //定义新用户
        Manage obj = new Manage();          //定义管理类
        obj.regist(c) ;                     //增加新用户

        obj.phone("138409025");obj.phone ("13840908925");     //该手机号申请打两
                                                              //次电话
        obj.save("138409025", 3);           //为该手机用户存钱
        obj.phone ("138409025");obj.phone ("138409025");      //该手机号申请打两
                                                              //次电话
    }
}
```

从 Manage 类、Test 类编码可知：对状态模式而言，某些情况下是需要一个上下文对象的集合管理类的，这样状态模式才显得更完备。

（6）使用标记变量改进 Context 类。

有一点还需要补充说明：上下文 CellContext 类 phone()方法是根据余额确定手机状态的。

从表意角度来说并不好，更好的方法是增加一个标识变量，方法如下。

```
class CellContext{
//仅列出了新增的代码，其他同上文中 CellContext 类中代码一致
    public final int NORMAL_STATE=1;          //正常态常量标识
    public final int OVERDRAW_STATE=2;        //欠费态常量标识
    public final int STOP_STATE=3;            //停止态常量标识
    public int getMark(){                     //根据余额返回当前状态标识
        int mark = 0;
        if(price > ICellState.NORMAL_LIMIT)
            mark = NORMAL_STATE;
        else if(price < ICellState.STOP_LIMIT)
            mark = STOP_STATE;
        else
            mark = OVERDRAW_STATE;
        return mark;
    }
    public boolean call(){
        int mark = getMark();
        ICellState state = null;
        switch(mark){
        case NORMAL_STATE:          //正常态
            state = new NormalState();break;
        case OVERDRAW_STATE:        //欠费态
            state = new OverDrawState();break;
        case STOP_STATE:            //停止态
            state = new StopState();break;
        }
        state.phone(this);
        return true;
    }
}
```

2. 利用具体状态类控制状态

仍以手机应用为例讲授使用具体的状态类来控制状态，具体代码如下。

（1）定义手机状态接口 ICellState。

```
interface ICellState{ /*同前例*/
    public float NORMAL_LIMIT = 0;          //正常状态
    public float STOP_LIMIT = -1;           //停机状态
    public float COST_MINUTE = 0.20f;       //话费标准
    public boolean phone(CellContext ct);
}
```

（2）定义手机用户三种状态类。

```
//正常状态下打电话类
class NormalState implements ICellState{
    public boolean phone(CellContext ct) {
        System.out.println(ct.name +":手机处于正常状态");
        int minute = (int)(Math.random()*10+1);
        ct.cost(minute);
        ct.setState();   //此处关键，重新设置打电话后的状态
        //保存信息到数据库的正常表中
```

```
            return false;
        }
    }
    //透支状态下打电话类
    class OverDrawState implements ICellState{
        public boolean phone(CellContext ct) {
            System.out.println(ct.name +":已处于欠费状态，请及时缴费");
            int minute = (int)(Math.random()*10+1);
            ct.cost(minute);
            ct.setState();    //此处关键，重新设置打电话后的状态
            //保存信息到数据库的透支表中
            return false;
        }
    }
    //停机类
    class StopState implements ICellState{
        public boolean phone(CellContext ct) {
            System.out.println(ct.name +":已处于停机状态，请及时缴费");
            //保存信息到数据库
            return false;
        }
    }
```

三个状态实现类与前面的例子相似，不同处在于 NormalState、OverDrawState 对象中打完电话后调用 setState()方法重新设置手机用户状态。由于在停机状态时不能打电话，因此在 StopState 类中不必调用 setCellState()方法，状态不变。

（3）手机上下文状态类 CellContext。

```
class CellContext2{
    public final int NORMAL_STATE=1;
    public final int OVERDRAW_STATE=2;
    public final int STOP_STATE=3;
    String strPhone;
    String name;
    float price;
    int mark = -100;            //初始化默认手机状态
    ICellState state;           //多态手机状态对象
    public CellContext2(String strPhone, String name, float price){
        this.strPhone = strPhone; this.name = name; this.price = price;
    }
    public int getMark(){
        int mark = 0;
        if(price > ICellState.NORMAL_LIMIT)
            mark = NORMAL_STATE;
        else if(price < ICellState.STOP_LIMIT)
            mark = STOP_STATE;
        else
            mark = OVERDRAW_STATE;
        return mark;
    }
    public void setState(){
        int curMark = getMark();
        if(curMark == mark)
            return;
```

```
            mark = curMark;
            switch(mark){
            case NORMAL_STATE:
                state = new NormalState();break;
            case OVERDRAW_STATE:
                state = new OverDrawState();break;
            case STOP_STATE:
                state = new StopState();break;
            }
        }
        public void save(float price){
            this.price += price;
        }
        public void cost(int minute){
            this.price -= ICellState.COST_MINUTE*minute;
        }
        public boolean call(){
            this.setState();
            state.phone(this);
            return true;
        }
    }
```

setState()方法设置当前手机用户状态，算法主要原理是：根据余额获取当前状态标识curMark，与前一状态标识 mark 进行对比，若两者相等，表明是同一状态，则返回；否则根据 mark 标识确定当前手机用户所处状态。

本例中实时设置状态有两种情况：第一种情况是打完电话后进行设置，这在三个具体状态类中已经进行描述了；第二种情况是存款后进行设置，也就是说，存款后要注意调用一次setState()方法。

8.4 应 用 探 究

【例 8-1】 计算机内存监控程序。

设计算机物理总内存为 total，空闲内存为 free，则有公式 ratio=free/total，表示内存空闲率。设两个阈值为 high、mid，high>mid。若 ratio≥high，空闲率相当高，表明内存处于“充裕”状态；若 mid≤ratio<high，空闲率正常，表明内存处于“良好”状态；若 ratio<mid，空闲率低，表明内存处于“紧张”状态。

先看程序执行界面，如图 8-3 所示。

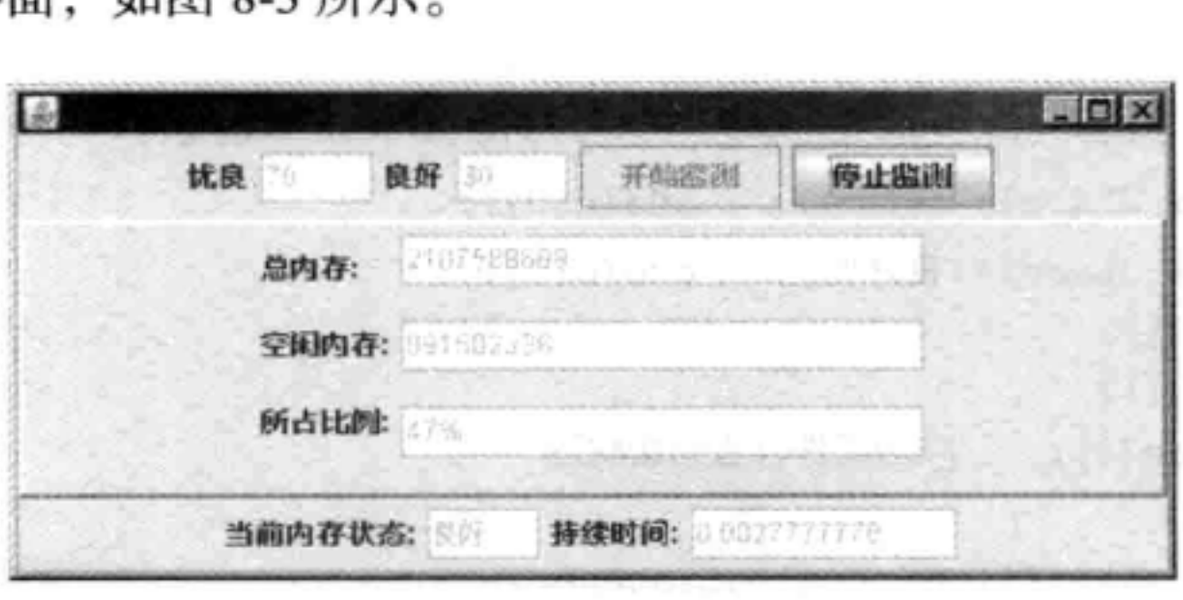


图 8-3 内存监测界面图

程序主要完成以下功能：①界面上可以输入阈值 high，low，有“开始监测”、“停止监测”按钮；②按字节显示总内存、空闲内存大小，显示空闲率；③显示当前内存状态，以及此状态的持续时间，单位是时。

很明显，界面由 3 个子面板组成：上方的参数控制面板，中间的数值显示面板，下方的状态面板。因此主要完成这三个类，再加上主窗口类共 4 个类的编制即可。下面逐一说明。

1. 参数控制面板类 CtrlPanel

```
class CtrlPanel extends JPanel{
    JComponent c[] = {new JTextField(4),new JTextField(4),
            new JButton("开始监测"),new JButton("停止监测")};
boolean bmark[][] = {{true,true,true,false},
            {false,false,false,true}};
ActionListener startAct = new ActionListener(){ //“开始监测”按钮响应
    public void actionPerformed(ActionEvent e){
        setState(1);                          //设置组件使能状态
        int high = Integer.parseInt(((JTextField)c[0]).getText());    //取高
                                                                      //阈值
        int low = Integer.parseInt(((JTextField)c[1]).getText());     //取低
                                                                      //阈值

        Container c = CtrlPanel.this.getParent();          //获得父窗口
        String className = c.getClass().getName();
        while(!className.equals("test4.MyFrame")){
            c = c.getParent();
            className = c.getClass().getName();
        }
        ((MyFrame)c).startMonitor(high, low);          //通知父窗口，开始监测
    }
};
ActionListener stopAct = new ActionListener(){         //“停止监测”按钮响应
    public void actionPerformed(ActionEvent e){
        setState(0);                                  //改变组件使能状态
        Container c = CtrlPanel.this.getParent();
        String className = c.getClass().getName();
        while(!className.equals("test4.MyFrame")){
            c = c.getParent();
            className = c.getClass().getName();
        }
        ((MyFrame)c).stopMonitor();                   //通知父窗口，停止监测
    }
};
public CtrlPanel(){                                   //向参数面板中添加组件
    add(new JLabel("优良"));   add(c[0]);
    add(new JLabel("良好"));   add(c[1]);
    add(c[2]);
    add(c[3]);
    setState(0);    //为组件设置初始状态

    ((JButton)c[2]).addActionListener(startAct);     //“开始监测”按钮事件注册
    ((JButton)c[3]).addActionListener(stopAct);      //“停止监测”按钮事件注册
```

```
        }
        void setState(int nState){
            for(int i=0; i<bmark[nState].length; i++){  //length值为2
                c[i].setEnabled(bmark[nState][i]);
            }
        }
}
```

本部分解决了组件的状态切换问题。初始时要求两个编辑框、“开始监测”按钮是使能的，“停止监测”按钮是禁止的；当单击“开始按钮”后，“停止监测”按钮变为使能状态，两个编辑控件、“开始监测”按钮变为禁止状态。也就是说，组件组共有两种状态，而且每种状态下组件组包含的各组件状态是一定的。如果按标准状态模式进行编程，就要编制两个具体状态类，很明显是比较累赘的。好的处理方式是：①定义组件组的固定二维状态布尔数组，每一行代表组件组的一个使能状况，如 bmark 数组；②定义组件组数组，均从 JComponent 类派生，如 c 数组；③利用循环，为组件组各组件设置使能状态，如 setState()方法。

2. 中间数值显示面板类 ContentPanel

```
class ContentPanel extends JPanel{
    JTextField totalField = new JTextField(20);      //总内存显示框
    JTextField freeField = new JTextField(20);       //空闲内存显示框
    JTextField ratioField = new JTextField(8);       //空闲率显示框
    public ContentPanel(){
        totalField.setEnabled(false);
        freeField.setEnabled(false);
        ratioField.setEnabled(false);
        Box b1 = Box.createVerticalBox();
        b1.add(new JLabel("总内存:")); b1.add(b1.createVerticalStrut(16));
        b1.add(new JLabel("空闲内存:")); b1.add(b1.createVerticalStrut(16));
        b1.add(new JLabel("所占比例:")); b1.add(b1.createVerticalStrut(16));
        Box b2 = Box.createVerticalBox();
        b2.add(totalField); b2.add(b2.createVerticalStrut(16));
        b2.add(freeField); b2.add(b2.createVerticalStrut(16));
        b2.add(ratioField); b2.add(b2.createVerticalStrut(16));

        add(b1); add(b2);
        setBorder(new BevelBorder(BevelBorder.RAISED));
    }
    public void setValue(long total,long free,int ratio){    //由主窗口MyFrame调用
        totalField.setText(""+total);
        freeField.setText(""+free);
        ratioField.setText("" +ratio+ "%");
    }
}
```

该类比较简单，一方面利用 Box 盒式布局生成界面，另一方面设置了界面显示填充方法 setValue()。

3. 状态面板类

内存有三种状态：充裕、良好、紧张，而且还要显示状态持续的时间。因此该部分功能用标准状态模式完成，具体如下。

（1）定义状态接口 IState。

```
interface IState{
    String getStateInfo();
    int getStateInterval();
}
```

getStateInfo()返回具体状态对象当前系统内存状态的标识串，为“充裕”、“良好”、“紧张”之一。本示例要求每 1 秒监测内存一次，因此，通过 getStateInterval()返回监测内存的次数，就知道该内存状态下的持续时间。

（2）三个具体状态实现类。

```
class HighState implements IState{          //内存充裕状态
    private int times;                      //监测次数
    public String getStateInfo() {
        return "充裕";
    }
    public int getStateInterval() {
        return times ++;
    }
}
class MidState implements IState{           //内存良好状态
    private int times;                      //监测次数
    public String getStateInfo() {
        return "良好";
    }
    public int getStateInterval() {
        return times ++;
    }
}
class LowState implements IState{           //内存紧张状态
    private int times;                      //监测次数
    public String getStateInfo() {
        return"一般";
    }
    public int getStateInterval() {
        return times ++;
    }
}
```

也许有读者会说：这三个类计数算法一致，仅反映状态信息的字符串不同，用上文 CtrlPanel 类中所述数组方法来控制状态，不是更简捷吗？其实还是有差别的。在 CtrlPanel 类中，状态仅涉及组件组各组件的使能状态，状态内容简单且固定，用数组进行控制状态是一个好的方法选择。而本示例中，三个具体状态类目前内容是简单的，但实际中不一定不会增加，比如当内存处于“紧张”态时，增加向责任人发短信功能，这时在派生状态类中增加相应功能就比较方便。因此，当状态变化简单、固定时，控制状态采用数组方法；当状态可能随着需求分析变化而变化时，控制状态采用标准状态模式方法更好。

（3）定义状态上下文类 StatePanel。

它也是状态面板类，与状态上下文类是同一个类。代码如下。

```
class StatePanel extends JPanel{
    JTextField txtInfo = new JTextField(4);
    JTextField txtHour = new JTextField(10);
    IState state;          //定义多态状态接口
```

```
int mark = -1;
public StatePanel(){        //向状态面板添加组件
    add(new JLabel("当前内存状态:")); add(txtInfo);
    add(new JLabel("持续时间:")); add(txtHour);
    txtInfo.setEnabled(false);
    txtHour.setEnabled(false);
}
public void setState(int mark){
    if(this.mark == mark) //内存状态不变
        return;
    this.mark = mark;
    //内存状态变化，则重置状态对象
    switch(mark){
    case 1:
        state = new HighState(); break;
    case 2:
        state = new MidState(); break;
    case 3:
        state = new LowState(); break;
    }
}
public void process(){
    txtInfo.setText(state.getStateInfo());
    int size = state.getStateInterval();   //getStateInterval()在不同状态下实现
    txtHour.setText("" + (float)size/3600);
}
}
```

每次进行内存监测时，都要调用 setState()方法确定内存状态变化与否，当上次监测成员变量 mark 值与传入形参值一定时，表明内存状态不变，则返回。否则，重新置内存具体状态对象。

4. 主窗口类 MyFrame

```
import java.lang.management.ManagementFactory;
import com.sun.management.OperatingSystemMXBean;
//import ……
class MyFrame extends JFrame implements ActionListener{
    CtrlPanel ctrlPanel = new CtrlPanel();                  //参数面板
    ContentPanel contentPanel = new ContentPanel();         //数值显示面板
    StatePanel statePanel = new StatePanel();               //状态面板
    Timer timer = new Timer(1000, this);                    //定时器，间隔时间 1 s
    int high, mid;                                          //高、低阈值
public void init(){
    add(ctrlPanel, BorderLayout.NORTH);
    add(contentPanel, BorderLayout.CENTER);
    add(statePanel, BorderLayout.SOUTH);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(500,220);
    setVisible(true);
}
public void startMonitor(int high, int mid){    //启动监测
    this.high = high; this.mid = mid;           //设置阈值
    timer.start();                              //启动定时器
```

```
    }
    public void stopMonitor(){                          //停止监测
        timer.stop();
    }
    public void actionPerformed(ActionEvent arg0) { //定时响应方法
        OperatingSystemMXBean osmxb =
(OperatingSystemMXBean)ManagementFactory.getOperatingSystemMXBean();
        long total = osmxb.getTotalPhysicalMemorySize();     //获取总物理内存
        long free = osmxb.getFreePhysicalMemorySize();       //获取空闲物理内存
        int ratio = (int)(free*100L/total);                  //计算空闲率
        contentPanel.setValue(total, free, ratio);           //设置数值显示面板

        int mark = -1;                                       //计算内存标识变量值
        if(ratio>=high)
            mark = 1;
        else if(ratio<mid)
            mark = 3;
        else
            mark = 2;

        statePanel.setState(mark);                           //为状态面板设置具体状态对象
        statePanel.process();                                //状态面板处理过程
    }
}
```

本类是流程控制类。主要过程如下：当单击“开始监测”按钮时，响应 startMonitor()方法，启动定时器。每隔 1 秒钟，执行定时响应方法 actionPerformed()。然后，获取总内存、空闲内存数据，送往数值显示面板 ContentPanel，状态面板 StatePanel 完成相关显示工作。

Java 平台提供了一些接口用于获得内存信息或操作内存。如虚拟机内存系统接口 MemoryMXBean，虚拟机内存管理器接口 MemoryManagerMXBean，虚拟机中的内存池接口 MemoryPoolMXBean，操作系统信息接口 OperatingSystemMXBean。由于本示例监测的是操作系统的内存，所以用到了 OperatingSystemMXBean 接口，通过 ManagementFactory 类中的静态方法 getOperatingSystemMXBean()，可以获得该接口示例的一个引用。

【例 8-2】 Web 学生成绩录入功能。

先看示例界面，在 IE 地址栏中输入 http://localhost:8080/StateJSP/gradeservlet?mark=0，界面如图 8-4 所示；录入成绩后，执行“保存数据”命令；单击“提交审核”按钮后，界面如图 8-5 所示。

图 8-4　初始录入数据界面

图 8-5 提交审核后界面

可以看出，数据处理有三种状态：

① 直接显示，表格中数据可以编辑；

②“保存数据”后继续显示，表格中数据仍可编辑；

③“提交审核”后，数据仍然显示，表格数据不可编辑，而且控制界面发生变化。

本例主要就是使用状态模式解决这些问题，涉及的数据库表说明如表 8-1 所示。

表 8-1 数据库表说明

序号	字段名	类型	关键字	描述
成绩状态表-examstate				
1	Examno	字符串	√	考试编号
2	Examstate	字符串		数据审核否？y:审核；n:未审核
成绩表-grade				
1	Examno	字符串	√	考试编号
2	Studno	字符串		学生编号
3	Studname	字符串		学生姓名
4	Grade	整形		成绩

为了简化问题规模，作以下假设：

① 成绩状态表 examstate 内容已设置，考试编号固定为“1000”，记录为（“1000”，“n”），表明未审核；

② 成绩表 grade 数据已导入，学生信息“考试编号”、“学生编号”、“学生姓名”存在，仅成绩未填充，因此本示例中的“保存”功能实际应是“更新”功能。具体代码如下。

1. 定义抽象状态 AbstractState

```
public abstract class AbstractState {
    protected String strSQL;
    public void setStrSQL(String s){
        strSQL = s;
    }
    public String getTableUI(){
        String s = "";
```

```
try{
    DbProc dbobj = new DbProc();
    Connection conn = dbobj.connect();
    Statement stm = conn.createStatement();
    ResultSet rst = stm.executeQuery(strSQL);
    s = "<div><table id='studtab' border='1'>";
    s += "<tr>"+
            "<th width='20'>序号</th><th width='100'>学号</th>"+
            "<th width='60'>姓名</th><th width='100'>成绩</th>"+
        "</tr>";
    int pos = 1;
        while(rst.next()){
            s += "<tr>" +
                "<td>"+pos+"</td>"+
                "<td>"+rst.getString("studno")+"</td>"+
                "<td>"+rst.getString("studname")+"</td>"+
                "<td>"+rst.getString("grade")+"</td>"+
                "</tr>";
            pos ++;
        }
        s += "</div></table>";
        rst.close(); stm.close(); conn.close();
    }
    catch(Exception e){e.printStackTrace();}
    return s;
}
public abstract void service(HttpServletRequest req, HttpServletResponse rep)
throws ServletException, IOException;
}
```

由上文可知，三种具体状态均有相同的表格显示，它们对应相同的查询 SQL 语句、绘制表格过程相同，因此基类中定义了与之对应的成员变量 strSQL 及绘制表格方法 getTableUI()。也使我们懂得，抽象状态不一定总用接口定义，有时也可能由抽象类表达。

2. 定义三种具体状态

仔细分析一下三种状态的关系："保存数据"状态与"初始显示"状态相比，增加一个保存功能，要将数据保存到成绩表 grade 中；"提交审核"状态与"保存数据"状态相比，增加一个"审核标识"保存功能，除了将数据保存到 grade 表中，还应更新成绩状态表 examstate，将考试编号为"1000"的记录 examstate 字段对应值设置为"y"，表明审核完成，以后数据就不能修改了。因此这三种状态的关系是继承的，而不是平行的。具体代码如下。

（1）直接显示状态类。

```
public class ShowState extends AbstractState {
protected String getCtrlUI(){
    String s = "<div>" +
        "<input type='submit' value='保存数据' id='save'/>" +"   &
        nbsp"+
        "<input type='submit' value='提交审核' id='check'/>" + "</div>";
    return s;
}
protected String getHidden(){
      String s ="<script type=\"text/javascript\" src=\"grade.js\"></script>\r\n"+
        "<form id='studform' action='gradeservlet' method='post'>" +
            "<input type='hidden' id='studno' name='studno'/>" +
```

```
                "<input type='hidden' id='studgrade' name='studgrade'/>" +

                "<input type='hidden' id='mark' name='mark'/>" +
            "</form>" + "<input type='hidden' id='checkvalue' value='n'/>";
        return s;
    }
    public void service(HttpServletRequest req, HttpServletResponse rep)
            throws ServletException, IOException {
        String s = getHidden();
        s += getCtrlUI();
        s += "<hr></hr>";
        s += getTableUI();
        rep.getWriter().print(s);
    }
}
```

getCtrlUI()比较简单，功能是生成控制界面 HTML 字符串。主要理解 getHidden()方法的作用，它起到方便客户信息传送到服务器端的作用。学生成绩信息是通过表格输入的，但 form 表单不支持表格数据传输，因此，必须在客户端把数据截获，填充到已知 form 的 input 标签对象中，由于无需显示，故把 input 标签置为 hidden 属性。通过该方法可知，学生的学号、成绩信息是通过 id 为 studform 的表单，name 分别为 studno、studgrade 的 input 标签传递到服务器端的。另外还要把状态标识 mark 传送到服务器端，服务器端根据此状态标识判断到底选择哪一个具体状态，从而完成相应功能。

（2）“保存”状态类 SaveState。

```
public class SaveState extends ShowState {
    protected void saveData(HttpServletRequest req, HttpServletResponse rep){//保存
                                                                    //数据到数据库
        try{
            String s = req.getParameter("studno");
            String strNO[] = s.split("-");
            s = req.getParameter("studgrade");
            String strGrade[] = s.split("-");
            DbProc dbobj = new DbProc();
            Connection conn = dbobj.connect();
            Statement stm = conn.createStatement();
            for(int i=0; i<strNO.length; i++){
                String strSQL = "update grade set grade=" +strGrade[i]+
                        " where examno='1000' and studno='" +strNO[i]+ "'";
                System.out.println(strSQL);
                stm.executeUpdate(strSQL);
            }
            stm.close(); conn.close();
        }
        catch(Exception e){e.printStackTrace();}
    }
    public void service(HttpServletRequest req, HttpServletResponse rep)
            throws ServletException, IOException {
        saveData(req, rep);
        super.service(req, rep);
    }
}
```

学号是以形如“10000-10001-10002”形式传到服务器端的，因此必须按“-”拆分字符串，

获取学生学号数组 strNO[]；成绩是以形如“90-80-70”形式传到服务器端的，因此必须按“-”拆分字符串，获取学生成绩数组 strGrade[]。

（3）“审核”状态类 CheckState。

```
public class CheckState extends SaveState {
    protected String getCtrlUI(){
        String s = "<div>" +
                   "该数据已经审核,不能修改"+
                   "</div>";
        return s;
    }
    protected String getHidden(){
        return"<input type='hidden' id='checkvalue' value='y' />";
    }
    protected void saveData(HttpServletRequest req, HttpServletResponse rep){
        super.saveData(req, rep);       //保存成绩信息
        try{                            //设置审核标志
        DbProc dbobj = new DbProc();
        Connection conn = dbobj.connect();
        Statement stm = conn.createStatement();
        String strSQL = "update examstate set examstate='y' where examno='1000'";
        stm.executeUpdate(strSQL);
        stm.close(); conn.close();
        }
        catch(Exception e){e.printStackTrace();}
    }
    public void service(HttpServletRequest req, HttpServletResponse rep)
            throws ServletException, IOException {
        saveData(req, rep);
        String s = getCtrlUI();
        s += "<hr></hr>";
        s += getTableUI();
        s += getHidden();
        rep.getWriter().print(s);
    }
}
```

3. 定义上下文状态控制类 GradeServlet

该类属于 servlet 类，URL 为“gradeservlet”。

```
public class GradeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public GradeServlet() {}
    protected void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
request.setCharacterEncoding("utf-8");
response.setContentType("text/html; charset=utf-8");

String examno = "1000";
String strMark = request.getParameter("mark");
int mark = Integer.parseInt(strMark);
try{     //确定是否已审核
    DbProc dbobj = new DbProc();
    Connection conn = dbobj.connect();
    Statement stm = conn.createStatement();
    String strSQL = "select * from examstate where examno='" +examno+ "'";
```

```
ResultSet rst = stm.executeQuery(strSQL);
rst.next();
if(rst.getString("examstate").equals("y")) //若已经审核
    mark = 2;     //则不论初始 mark 值为多少，都置为审核状态
rst.close();stm.close();conn.close();

AbstractState state = null;
switch(mark){
case 0:  //显示状态
    state = new ShowState(); break;
case 1:  //保存状态
    state = new SaveState(); break;
case 2:  //审核状态
    state = new CheckState();break;
}
state.setStrSQL("select * from grade where examno='"+examno+"'");
state.service(request, response);
}
catch(Exception e){
    e.printStackTrace();
}
}
}
```

到此为止，基本功能类编制完毕。仔细分析会发现，并没有对表格是否可编辑加以控制，也没有对“保存数据”、“提交审核”按钮进行消息注册，更没有对如何获取表格数据加以描述。这些功能都是由 JavaScript 代码完成的，代码如下。

4. JavaScript 功能代码，保存在文件 grade.js

```
window.onload = init; //当页面加载后，调用 init 方法
function addEvent(obj, type, fn){
    obj.attachEvent("on"+type, fn);
}
function init(){       //给各组件加消息映射
    var saveobj = document.getElementById("save");
    var checkobj = document.getElementById("check");
    var tabobj = document.getElementById("studtab");
    var check = document.getElementById("checkvalue");
    if(check.value != "y")
        addEvent(tabobj,"click",editProc);
    addEvent(saveobj,"click",saveProc);     // “保存”按钮消息响应方法是 saveProc()
    addEvent(checkobj,"click",checkProc)    // “审核”按钮消息响应方法是 checkProc()
}
function getData(){        //获取表格数据
    var tabobj = document.getElementById("studtab");
    var studno = "";       //学号信息字符串
    var studgrade = "";    //成绩信息字符串
    for(var i=1; i<tabobj.rows.length; i++){
        studno += tabobj.rows[i].cells[1].innerText+ "-";
        studgrade += tabobj.rows[i].cells[3].innerText+ "-";
    }
    document.getElementById("studno").value = studno;
    document.getElementById("studgrade").value = studgrade;
```

```
}
function saveProc(e){        //“保存”按钮响应方法
    getData();
    document.getElementById("mark").value = "1";
    var studobj = document.getElementById("studform");
    studobj.submit();        //信息传送到服务器端
}
function checkProc(e){           //“审核”按钮响应方法
    getData();
    document.getElementById("mark").value = "2";
    var studobj = document.getElementById("studform");
    studobj.submit();        //信息传送到服务器端
}
function editProc(e){        //表格编辑处理
    var obj = e.target || window.event.srcElement;
    if(obj.tagName != "TD")
        return;
    var trobj = obj.parentElement;
    if(trobj.cells[3] == obj)//若是表格第 4 列——成绩
        entryeditcell(obj);//则进行编辑处理
}
function entryeditcell(obj){
    var w = obj.offsetWidth-4;
    var subs = "style='border-style:none;width:" +w+ "px;'";
    obj.innerHTML="<input id='myedit' type='text' " +subs+ " value="+obj.innerText+"
onblur='leaveeditcell(this)'>";//插入文本框，且指定内容
    myedit.focus();
}
function leaveeditcell(obj){
    var tdobj = obj.parentElement;
    tdobj.innerText = obj.value;
}
```

第9章　访问者模式

9.1　问题的提出

人们认识事物常常有一个循序渐进的过程，不可能是一蹴而就的。例如某事物经分析后有功能1、功能2，但是或者随着时间的推移，或者随着需求分析的变化，或者随着二次开发的需要，还必须完成功能 3。计算机如何能更好地描述这样的特点呢？可能有读者认为这非常简单，类似表 9-1 实现就可以了。

表 9-1　　原功能与变化后功能对比表

原功能	变化后功能

interface IFunc{
　　void func();　　//功能1
　　void func2();　　//功能2
}
class Thing implements IFunc{
　　public void func(){}
　　public void func2(){}
}
	interface IFunc{
	　　void func();　　//功能1
	　　void func2();　　//功能2
	　　void func3();　　//功能3
	}
	class Thing implements IFunc{
	　　public void func(){}
	　　public void func2(){}
	　　public void func3(){}
	}

可以看出，新增功能3主要是通过在IFunc接口中增加方法定义func3()，再在实现类Thing中重写 func3()方法完成的。这种方法是常规思路，最大特点是若增加新功能，必须修改接口及相应实现。那么，能不能不修改 IFunc、Thing，也能实现新增的功能 3 呢？访问者模式是具体实现的手段之一。

9.2　访问者模式

访问者模式的目的是封装一些施加于某种数据结构元素之上的操作，一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。为不同类型的元素提供多种访问操作方式，且可以在不修改原有系统的情况下增加新的操作方式，这就是访问者模式的模式动机。

考虑这样一个应用：已知三点坐标，编写功能类，求该三角形的面积和周长。

如果采用访问者模式架构，应当这样思考：目前已确定的需求分析是求面积和周长功能，但有可能将来求三角形的重心、垂心坐标，内切、外界圆的半径等，因此，在设计时必须考虑如何屏蔽这些不确定情况。具体代码如下。

1. 定义抽象需求分析接口 IShape

```
interface IShape{
    float getArea();                    //明确的需求分析
    float getLength();                  //明确的需求分析
    Object accept(IVisitor v);          //可扩展的需求分析
}
```

着重理解可扩展的需求分析方法 accept()，它在形式上仅是一个方法，但是按访问者模式而言，它却可以表示将来可以求重心、垂心坐标等功能，是一对多的关系，因此 IVisitor 一般来说是接口或抽象类，“多”项功能一定是由 IVisitor 的子类来实现的。那么为什么返回值是 Object 类型呢？可以这样理解，例如重心坐标由两个浮点数表示，外接圆半径由一个浮点数表示，为了屏蔽返回值差异，返回值定义成 Object，表明可以返回任意对象类型。

2. 定义具体功能实现类 Triangle

```
class Triangle implements IShape{
    float x1,y1,x2,y2,x3,y3;                //三角形三点坐标
    public Triangle(float x1,float y1,float x2,float y2,float x3,float y3){

        this.x1=x1;this.y1=y1;
        this.x2=x2;this.y2=y2;
        this.x3=x3;this.y3=y3;
    }
    public float getDist(float u1,float v1,float u2,float v2){   //求任意两点距离
        return (float)Math.sqrt((u1-u2)*(u1-u2)+(v1-v2)*(v1-v2));
    }
    public float getArea(){                 //固定需求分析求面积
        float a = getDist(x1,y1,x2,y2);
        float b = getDist(x1,y1,x3,y3);
        float c = getDist(x2,y2,x3,y3);
        float s = (a+b+c)/2;
        return (float)Math.sqrt(s*(s-a)*(s-b)*(s-c));//海伦公式求面积
    }
    public float getLength(){               //固定需求分析求周长
        float a = getDist(x1,y1,x2,y2);
        float b = getDist(x1,y1,x3,y3);
        float c = getDist(x2,y2,x3,y3);
        return a+b+c;
    }
    public Object accept(IVisitor v){  //可扩展需求分析
        return v.visit(this);
    }
}
```

着重理解 accept()方法，可以看出 IVisitor 接口中一定定义了多态方法 visit()，那为什么把 this 引用传过去呢？可以这样理解：例如求三角形重心坐标，它的功能一定是在 IVisitor 的子类实现的，那么该子类一定得知道三角形的三个顶点坐标，因此把 this 引用传过去，相当于 IVisitor 的子类可访问 Triangle 类的成员变量，编制求重心坐标就容易了。

3. 定义访问者接口 IVisitor

```
interface IVisitor{
    Object visit(Triangle t);
}
```

至此为止，有了 1、2、3 的代码，访问者模式的代码框架就已经构建起来了。如果需求分析没有变化，那么程序一直应用即可；如果需求分析发生变化，则基础功能类不用变化，只要定义 IVisitor 接口的具体功能实现类就可以了，例如求三角形重心坐标代码如下。

4. 定义重心坐标实现类 CenterVisitor

```
class Point{
    float x,y;
}
class CenterVisitor implements IVisitor{
    public Object visit(Triangle t){
        Point pt = new Point();
        pt.x = (t.x1+t.x2+t.x3)/3;
        pt.y = (t.y1+t.y2+t.y3)/3;
        return pt;
    }
}
```

一个简单的测试类如下。

```
public class Test3 {
    public static void main(String[] args) {
        IVisitor v = new CenterVisitor();          //定义求重心具体访问者对象
        Triangle t = new Triangle(0,0,2,0,0,2);    //定义三角形对象
        Point pt = (Point)t.accept(v);             //通过访问者对象求三角形重心坐标
        System.out.println(pt.x+"\t"+pt.y);
    }
}
```

可以知道，如果再想增加一个求三角形外接圆半径功能，只需再定义一个新类实现 IVisitor 接口，在该类中完成求外接圆半径功能即可。

总之，访问者模式的抽象类图如图 9-1 所示。

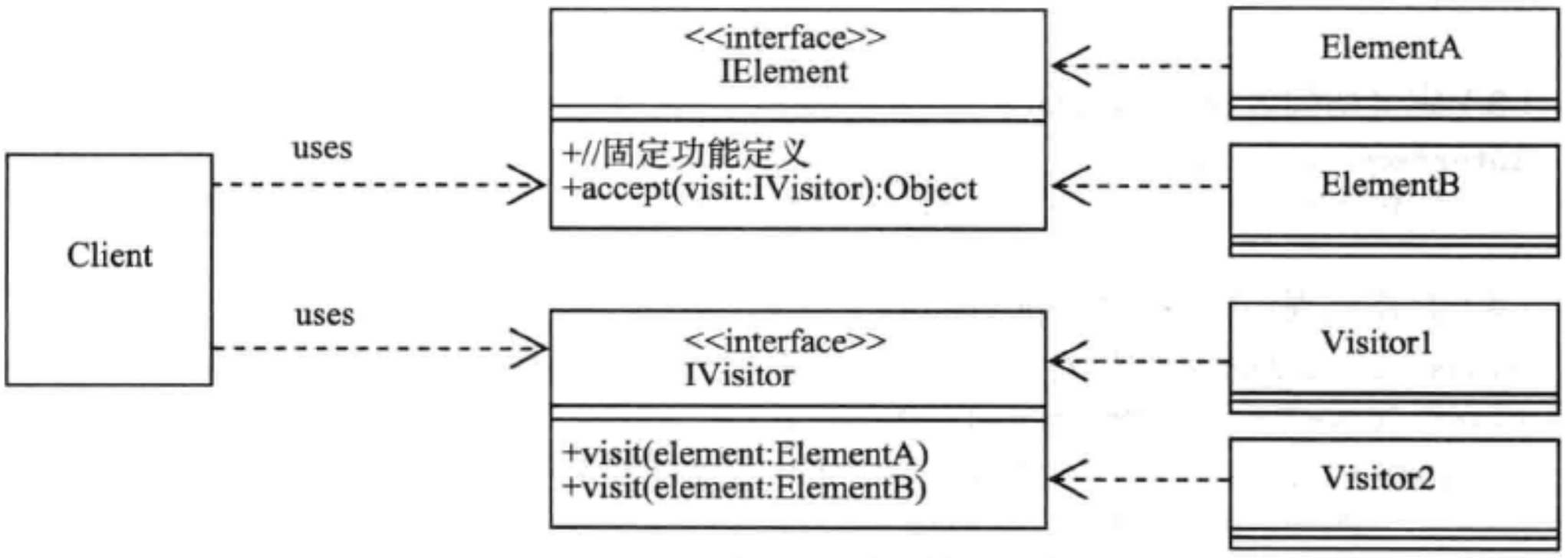


图 9-1　访问者模式抽象类图

该图与上文示例中稍有不同。该图有两个具体元素类（ElementA、ElementB），两个具

体访问者类（Visitor1、Visitor2），而示例中仅有一个元素类、一个访问者类，虽然个数不同，但编程框架是相似的，请读者自行完成。

访问者模式主要涉及以下四种角色。

- IElement：抽象的事物元素功能接口，定义了固定功能方法及可变功能方法接口。
- Element：具体功能的实现类。
- IVisitor：访问者接口，为所有访问者对象声明一个 visit 方法，用来代表为对象结构添加的功能，原则上可以代表任意的功能。
- Visitor：具体访问者实现类，实现要真正被添加到对象结构中的功能。

9.3 深入理解访问者模式

1. 应用反射技术

分析 9.2 节中示例可知，若三角形需要新增 *n* 个功能，则必须定义 *n* 个具体访问者类。毫无疑问，有时候这样的代码会显得非常臃肿，利用反射技术可以方便解决这一问题。例如，现在三角形需要新增计算重心坐标，及求三角形内切圆半径功能，具体代码如下。

（1）定义抽象需求分析接口 IShape。

```
interface IShape{
    float getArea();
    float getLength();
    Object accept(IVisitor v, String method);
}
```

与 9.2 节示例相比，仅可变方法 accept()有不同，增加了字符串变量 method，代表将要采用反射技术的方法名称。

（2）定义具体功能实现类 Triangle。

```
class Triangle implements IShape{
    //其余代码同 9.2 节
    public Object accept(IVisitor visitor, String method){
        return visitor.visit(this, method);
    }
}
```

（3）定义访问者接口 IVisitor。

```
interface IVisitor{
    Object visit(Triangle t, String method);
}
```

（4）具体实现访问者类 ShapeVisitor。

```
class Point{/*同 9.2 节*/}
class ShapeVisitor implements IVisitor{
    public Object getCenter(Triangle t){ //获取重心坐标
        Point pt = new Point();
        pt.x = (t.x1+t.x2+t.x3)/3;
        pt.y = (t.y1+t.y2+t.y3)/3;
        return pt;
    }
```

```
public Float getInnerCircleR(Triangle t){ //获取内切圆半径
    float area = t.getArea();
    float len = t.getLength();
    return new Float(2.0f*area/len);
}
public Object visit(Triangle t, String method){//访问者接口转发方法
    Object result = null;
    try{
        Class classinfo = this.getClass();
        Method mt = classinfo.getMethod(method,Triangle.class);
        result =  mt.invoke(this, new Object[]{t});
    }
    catch(Exception e){}
    return result;
  }
}
```

可以看出，访问者接口实现方法 visit()中运用了反射技术，visit()方法仅起到了一个转发作用，具体功能是由非多态方法 getCenter()及 getInnerCircleR()完成的。一个简单的测试类如下。

```
public class Test4 {
    public static void main(String[] args) throws Exception{
        IVisitor v = new ShapeVisitor();                    //定义访问者
        Triangle t = new Triangle(0,0,2,0,0,2);             //定义具体事物元素类
        Point pt = (Point)t.accept(v,"getCenter");    //获得重心坐标
        System.out.println("重心坐标 x="+pt.x+"\ty="+pt.y);
        Float f = (Float)t.accept(v, "getInnerCircleR");       //获得内切圆半径
        System.out.println("内切圆半径 r=" +f.floatValue());
    }
}
```

2. 抽象访问者定义形式

已知某元素有两种类型，定义访问者基本模式框架。代码简述如下所示。

（1）定义抽象事物 IElement。

```
interface IElement{
    public void accept(IVisitor v);          //访问者接口方法
}
```

（2）定义两个具体事物 Element、Element2。

```
class Element implements IElement{
    public void accept(IVisitor v){
        v.visit(this);
    }
}
class Element2 implements IElement{
    public void accept(IVisitor v){
        v.visit(this);
    }
}
```

（3）定义抽象访问者及具体实现类。

方法 1：仔细分析 Element、Element2 两个类中的 accept()方法，有一行重要的语句 v.visit(this)，this 代表 Element(或 Element2)对象。因此，抽象访问者定义成如下形式。

```
interface IVisitor{
    public void visit(Element obj);
    public void visit(Element2 obj);
}
```

也就是说，如果 IElement 有 *n* 个子类，抽象访问者接口就要定义 *n* 个方法。具体访问者直接实现该抽象接口即可。

方法 2：虽然上文 v.visit(this)代码中 this 代表 Element(或 Element2)对象，但它们都是 IElement 的子类对象。因此，抽象访问者定义成如下形式。

```
interface IVisitor{
    public void visit(IElement obj);
}
```

也就是说，不论 IElement 有多少个子类，抽象访问者接口仅定义一个方法。那么，在哪里判断 IElement 引用是哪个具体的子类对象呢？当然在访问者具体实现子类中，一个示例代码如下。

```
class OneVisitor implements IVisitor{
    public void visit(IElement obj){
        if(obj instanceof Element)
        {/*处理代码*/}
        else
        {/*处理代码*/}
    }
}
```

3. 构建集合对象自适应功能框架

自适应功能框架的含义是当需求分析发生变化后，仅添加相应的具体功能，而程序的主框架不变。集合是由元素组成的，集合和元素是相对的。例如，*A* 是 *B* 的集合，*B* 是 *C* 的集合，一般来说，只有 *A*、*B*、*C* 均是自适应的，整个系统才能称为功能框架。访问者模式是实现集合对象自适应功能框架的重要手段。

考虑一个存单管理系统：假设仅在中国工商银行和中国农业银行有定期存单，编制相应的自适应程序管理框架。

分析：存单管理系统固定（业务）功能不在本示例考虑范围内，仅考虑如何处理可变功能的实现方法。存在多家银行每个银行都能有多个存单，从这句话可得出三个重要的层次："存单、银行、银行组"，因此主要编制这三方面的自适应程序功能框架。具体代码如下。

（1）定义泛型访问者接口 IVisitor。

```
interface IVisitor<T>{
    void visit(T s);
}
```

（2）定义存单相关类。

因为有中国工商银行及中国农业银行两种存单，而且要求存单类具有自适应功能，因此定义了抽象基类 Sheet、工行存单子类 ICBCSheet、农行存单子类 ABCSheet。

```
//存单抽象基类 Sheet
abstract class Sheet{
    String account;      //账号
```

```
String name;            //姓名
float  money;           //余额
String startDate;       //存款日期
int range;              //期限
public Sheet(String account,String name,float money,String startDate,int
       range){
    this.account=account;this.name=name;this.money=money;
    this.startDate=startDate;this.range=range;
}
public void accept(IVisitor<Sheet> v){
        v.visit(this);
    }
}
```

accept()方法预留了与访问者 IVisitor 之间的通信接口，是实现存单自适应功能的根本所在。

```
//工行存单子类 ICBCSheet，农行存单子类 ABCSheet
class ABCSheet extends Sheet{     //农行存单类
    public ABCSheet(String account,String name,float money,String startDate,int
        range){
        super(account,name,money,startDate,range);
    }
}
class ICBCSheet extends Sheet{    //工行存单类
    public ICBCSheet(String account,String name,float money,String startDate,int
        range){
        super(account,name,money,startDate,range);
    }
}
```

（3）各银行存单管理类。

```
//抽象银行存单管理类 Bank
abstract class Bank{
    public void accept(IVisitor<Bank> v){
        v.visit(this);
    }
    public abstract void process(IVisitor<Sheet> visit);
}
```

本示例中银行是存单的集合。对银行组来说，它又是元素。也就是说，银行既是“集合”又是“元素”，那么银行类必须在这两方面实现自适应功能，accept()方法对应“元素”自适应功能，process()方法对应“集合”自适应功能。

```
//两个银行子类 ABCBank、ICBCBank
class ABCBank extends Bank{
    Vector<ABCSheet> v = new Vector();
    void add(ABCSheet s){
        v.add(s);
    }
    public void process(IVisitor<Sheet> visit){
        for(int i=0; i<v.size(); i++){
            v.get(i).accept(visit);
        }
```

```
    }
}
class ICBCBank extends Bank{
    Vector<ICBCSheet> v = new Vector();
    void add(ICBCSheet s){
        v.add(s);
    }
    public void process(IVisitor<Sheet> visit){
        for(int i=0; i<v.size(); i++){
            v.get(i).accept(visit);
        }
    }
```

每个银行子类中都定义了 Vector 类型的成员变量，表明是相应存单的集合。process()方法主要是利用循环，统一完成对各个具体存单的自适应功能。

（4）银行组存单管理类。

```
class BankGroup{
    Vector<Bank> v = new Vector();
    void add(Bank bank){
        v.add(bank);
    }
    public void accept(IVisitor<BankGroup> v){
        v.visit(this);
    }
    public void process(IVisitor<Bank> visit){
        for(int i=0; i<v.size(); i++){
            v.get(i).accept(visit);
        }
    }
}
```

本示例中银行组是银行的集合，它本身又是元素。也就是说，银行组既是“集合”又是“元素”，那么银行组类必须在这两方面实现自适应功能，accept()方法对应“元素”自适应功能，process()方法对应“集合”自适应功能。

有了（1）~（4）代码，自适应功能框架就基本构建完成了。通过上述代码的感性认识，得出更一般的结论：假设有元素 *A*、*B*、*C*、*D*、*E*，后者是前者的集合。由于 *A* 是最底层元素，所以该类一般定义成 abstract 类，里面含有 accept()方法，之后再定义 *A* 的各派生子类；*E* 是最顶层元素，向下有集合 *D*，所以该类一般定义成普通类，里面含有 accept()、process()方法；*B*、*C*、*D* 是中间元素，所以一般先定义一个 abstract 抽象基类，里面含有 accept()方法，抽象 process()方法，之后再定义各派生子类，重写 process()方法。

（5）简单的测试类编制方法。

一般来说，若需求分析无变化，则上述自适应框架不起作用；若需求分析发生变化，则要定义相应的访问者接口实现类及相关调用代码。例如，定义一个存单访问者及一个银行访问者的具体代码如下。

```
class SheetVisitor implements IVisitor<Sheet>{
    private void ABCProc(ABCSheet sheet){
        System.out.println("ABCSheet process");
    }
    private void ICBCProc(ICBCSheet sheet){
```

```
            System.out.println("ICBCSheet process");
        }
        public void visit(Sheet s) {
            if(s instanceof ABCSheet){
                ABCProc((ABCSheet)s);
            }
            if(s instanceof ICBCSheet){
                ICBCProc((ICBCSheet)s);
            }
        }
}
class BankVisitor implements IVisitor<Bank>{
    private void ABCProc(ABCBank bank){
        System.out.println("ABCBank process");
    }
    private void ICBCProc(ICBCBank bank){
        System.out.println("ICBCBank process");
    }
    public void visit(Bank b){
        if(b instanceof ABCBank){
            ABCProc((ABCBank)b);
        }
        if(b instanceof ICBCBank){
            ICBCProc((ICBCBank)b);
        }
    }
}
```

编制具体访问者代码的一般思路：在接口实现方法 visit()中利用 instanceof 关键字实现方法转发功能，在转发方法中完成具体的代码编制。

一个简单的测试代码如下。

```
public class Test {
    public static void main(String[] args) {
        ICBCSheet s = new ICBCSheet("1000","zhang",100,"2012-1-1",3);//定义一个
                                                                    //工行单据
        IVisitor v = new SheetVisitor();        //定义单据访问者
        s.accept(v);                            //对一个具体单据自适应

        ICBCBank manage = new ICBCBank();       //定义工行对象
        manage.add(s);                          //工行加一个单据
        manage.process(v);                      //对工行所有单据实现自适应功能
    }
}
```

9.4　应 用 探 究

【例 9-1】 学生成绩查询功能。

学生成绩已经保存在文本文件中，格式如表 9-2 所示。

表 9-2　　学生成绩文本文件格式说明

文本文件示例	文本文件说明
班级：一年一班 1001 zhang 70 80 90 1002 zhang2 71 81 91 1003 zhang3 69 79 89 1004 zhang4 65 80 90 班级:一年二班 1005 zhao 70 80 90 1006 zhao2 71 81 91	• 各班级均以“班级”为前缀，表示以下是该班成绩。 • 一行仅有一个学生记录，表示学号、姓名、语文、数学、外语成绩，中间由单空格隔开。 • 本示例共两个班的数据。

程序执行主界面如图 9-2 所示。

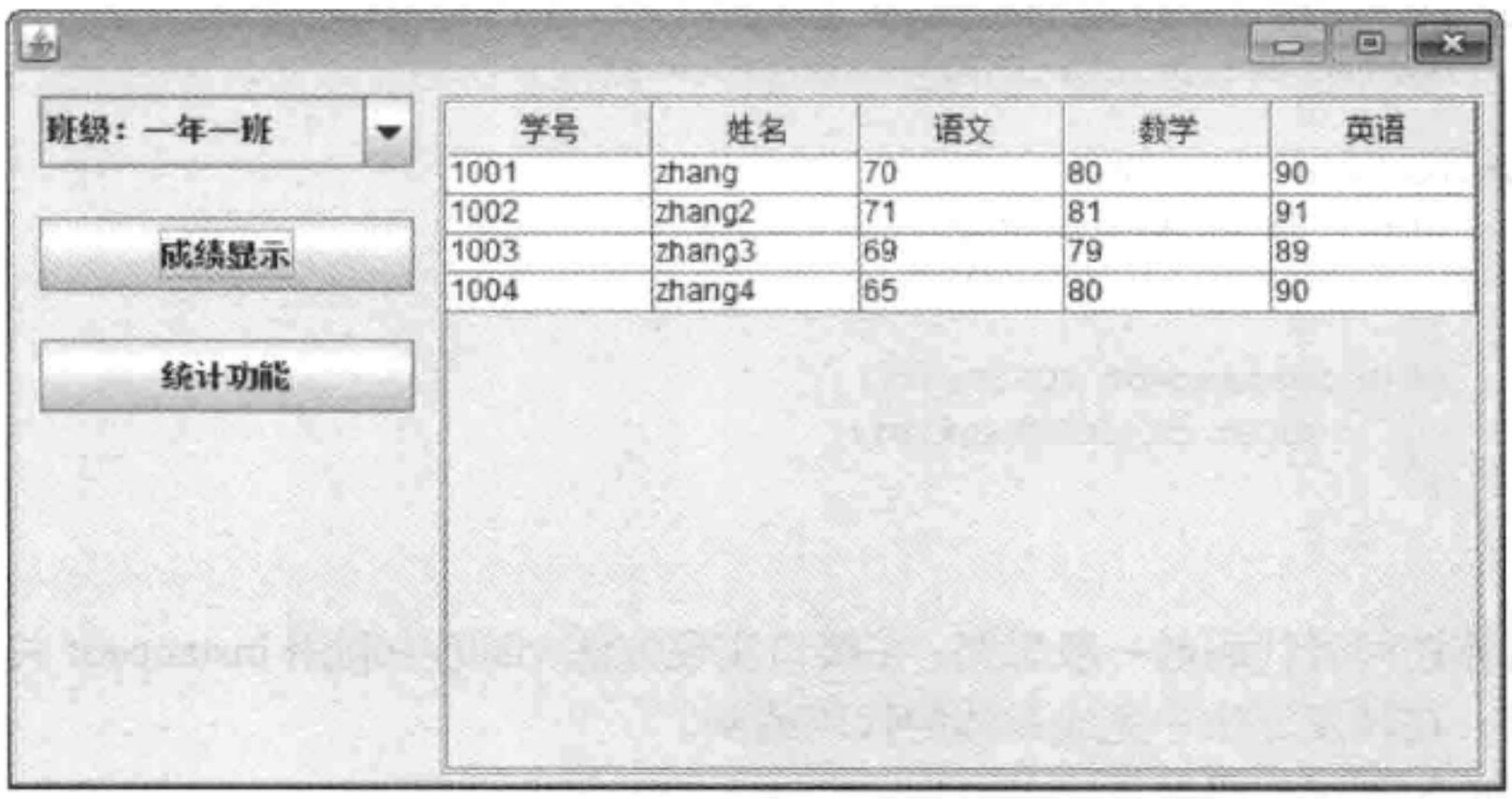


图 9-2　学生成绩查询功能主界面

学生成绩查询包括一些固定的功能，如成绩显示、排序等。为了程序简捷，本示例实现了成绩显示和成绩统计功能。在图 9-2 中，先通过下拉菜单选择班级，单击“成绩显示”按钮，则在右侧面板中以表格形式显示该班学生具体成绩信息内容。学生成绩查询还包括统计查询，该功能是用访问者模式实现的。在图 9-2 中，单击“统计功能”按钮，则在右侧面板中以表格形式显示该班学生总成绩分段信息情况。

本部分代码主要由三部分组成：学生成绩信息获得、界面生成、统计功能具体访问者。下面具体加以说明。

1. 学生成绩信息获得相关代码

学生的集合构成了班级，班级的集合构成了年级，学生类 Student、班级类 Banji、年级类 Grade 代码如下。

```
//学生类 Student
class Student{
    String studNO;    //学号
    String name;      //姓名
    int chinese;      //语文成绩
    int math;         //数学成绩
    int english;      //外语成绩
```

```
    public Student(String t[]){
        studNO = t[0]; name = t[1];
        chinese = Integer.parseInt(t[2]);
        math = Integer.parseInt(t[3]);
        english = Integer.parseInt(t[4]);
    }
}
//班级类 Banji
interface IVisitor{
    void visit(Banji obj, JPanel panel);
}
class Banji{
    Vector<Student> v = new Vector();                    //班级是学生的集合
    public Vector<Student> getV() {
        return v;
    }
    void add(Student s){                                 //班级增加学生
        v.add(s);
    }
    void statistics(IVisitor v, JPanel panel){           //访问者接口方法
        v.visit(this, panel);
    }
}
```

按照需求分析是按班级进行数据统计分析，因此在该类内应有访问者模式框架代码，statistics()方法体现了访问者模式思想，IVisitor 是抽象访问者。那么第二个参数 JPanel 类型形参 panel 是什么意思呢？可以这样理解，在具体访问者对象中进行运算后，要把结果体现在 panel 界面上，该 panel 即指图 9-2 界面中的右侧面板。

在 9.1 节中，标准访问者模式中 visit()方法一般仅有一个参数，而本例中则有两个参数。因此，一定要具体问题具体分析， 希望读者多加体会。

```
//年级类 Grade
class Grade{
    Map<String, Banji> map = new HashMap();   //key:班级名称
    public Banji add(String banji){
        Banji obj = new Banji();
        map.put(banji, obj);
        return obj;
    }
    public void readFile(String strFile){     //读学生成绩信息文件
String s = "";
Banji obj = null;
try{
    FileReader in = new FileReader(strFile);
    BufferedReader in2 = new BufferedReader(in);
    while((s=in2.readLine())!=null){
        s = s.trim();
        if(s.equals("")) continue;
        if(s.startsWith("班级")){
            obj = add(s);
            continue;
```

```
                }
                String t[] = s.split("");
                Student stud = new Student(t);
                obj.add(stud);
            }

            in2.close();
            in.close();
        }
        catch(Exception e){e.printStackTrace();}
    }
```

readFile()方法完成解析学生成绩文本文件，格式严格如表 9-2 所示，依次形成 Student->Banji->Grade 集合关系。

2. 界面生成相关代码

```
class MyFrame extends JFrame{
    Grade grade = new Grade();                          //定义年级成员变量
    JPanel contentPane = new JPanel();                  //定义主界面右侧内容面板
    JComboBox combo;                                    //主界面选择班级下拉框对象

    ActionListener a1 = new ActionListener(){   //“成绩显示”按钮响应方法
        public void actionPerformed(ActionEvent e){
            String factor  = (String)combo.getSelectedItem();
            Banji obj = (Banji)grade.map.get(factor);
            Vector<Student> v = obj.getV();
            String title[]={"学号","姓名","语文","数学","英语"};
            String d[][] = new String[v.size()][5];
            for(int i=0; i<v.size(); i++){
                Student unit = v.get(i);
                d[i][0]=unit.studNO; d[i][1]=unit.name; d[i][2]=""+unit.chinese;
                d[i][3]=""+unit.math; d[i][4]=""+unit.english;
            }
            showTable(d, title);
        }
    };
    ActionListener a2 = new ActionListener(){   //“统计功能”按钮响应方法
        public void actionPerformed(ActionEvent e){
            String factor  = (String)combo.getSelectedItem();
            Banji obj = (Banji)grade.map.get(factor);
            Vector<Student> v = obj.getV();
            obj.statistics(new StaVisitor(),contentPane);
        }
    };
    public void showTable(String d[][], String title[]){
        contentPane.removeAll();
        contentPane.setLayout(new BorderLayout());
        JTable tab = new JTable(d, title);
        JScrollPane scr = new JScrollPane(tab);
        contentPane.add(scr);
        contentPane.updateUI();
    }
    public void init(String strFile){
        grade.readFile(strFile);        //读数据文件，形成数据集合
```

```
        setLayout(null);                    //布局管理器设置为空
        Set<String> set = grade.map.keySet();
        Object info[] = set.toArray();
        combo = new JComboBox(info);    //填充班级下拉框内容
        combo.setBounds(10, 10, 150, 30);
        add(combo);                         //将班级下拉框添加到界面中

        JButton dispBtn = new JButton("成绩显示");
        JButton infoBtn = new JButton("统计功能");
        dispBtn.addActionListener(a1);      //注册“成绩显示”消息
        infoBtn.addActionListener(a2);      //注册“统计功能”消息
        dispBtn.setBounds(10, 60, 150, 30);
        infoBtn.setBounds(10,110, 150, 30);
        add(dispBtn);                       //将“成绩显示”按钮添加到界面上
        add(infoBtn);                       //将“统计功能”按钮添加到界面上

        contentPane.setBounds(170, 10, 420, 280);
        contentPane.setBorder(BorderFactory.createEtchedBorder());
        add(contentPane);                   //将“内容面板”添加到界面上

        this.setSize(600,320);
        this.setResizable(false);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
```

init()是初始化信息方法。包括：启动读学生成绩文本文件，形成数据集合；主界面布局管理器设置为空，添加各子控件及相关按钮的消息注册。主要有 4 个子组件，分别是班级下拉框、成绩显示、统计功能按钮、右侧内容面板。

“成绩显示”、“统计功能”按钮是由内部匿名类对象 a1、a2 响应并完成的，着重比较 actionPerformed()方法的不同。a1 中的 actionPerformed()方法内容是固定的，a2 中的 actionPerformed()方法内容是可变的，随着访问者接口对象的不同而不同。

3. 统计功能具体访问者代码

由于是语文、数学、外语三门成绩，本示例统计<200，200~210，210~220，…，290~300 各分数段的人数，具体代码如下。

```
class StaVisitor implements IVisitor{
    public void visit(Banji obj, JPanel panel){
        String title[] = {"成绩区间", "人数"};
        String d[][] = new String[11][2];
        d[0][0]="<200";
        for(int i=1; i<d.length; i++)
            d[i][0] = ""+(200+(i-1)*10) +"~"+ (200+(i*10));
        int num[] = new int[11];
        Vector<Student> v = obj.getV();
        for(int i=0; i<v.size(); i++){
            Student s = v.get(i);
            int total = s.chinese+s.math+s.english;
            if(total<200){ num[0]++; }
            else
```

```
            num[(total-200)/10 +1]++;
        }
        for(int i=0; i<num.length; i++){
            d[i][1] = ""+num[i];
        }

        panel.removeAll();
        panel.setLayout(new BorderLayout());
        JTable tab = new JTable(d, title);
        JScrollPane scr = new JScrollPane(tab);
        panel.add(scr);
            panel.updateUI();
        }
    }
```

4. 一个简单的测试类

```
public class Test {
    public static void main(String[] args) {
        new MyFrame().init("d:/stud.dat");
    }
}
```

5. 利用反射技术增强访问者调用功能

（1）定义访问者信息配置文件 info.xml，如表 9-3 所示。

表 9-3　　info.xml 定义

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="成绩分段统计">StaVisitor</entry>
</properties>
```

键表明访问者功能，值表示具体访问者类。可以按此格式，增加多个访问者配置信息。

（2）解析配置文件。

在 Grade 类中解析该配置文件，形成 map 键值映射对，键表示访问者功能，值表示具体访问者类。

```
class Grade{
    Map<String, String> xmlmap = new HashMap();
    public void readXMLFile(String strXMLFile){
        Properties p = new Properties();
        try{
            p.loadFromXML(new FileInputStream(strXMLFile));//装载配置文件
            Set s = p.keySet();
            Iterator<String> it = s.iterator();
            while(it.hasNext()){
                String key = it.next();
                xmlmap.put(key, p.getProperty(key));
            }
        }
        catch(Exception e){e.printStackTrace();}
    }
    //其余所有代码同上文例 9-1
}
```

（3）动态生成界面，利用反射技术调用具体访问者类。

```
class MyFrame extends JFrame{
    //其余所有定义同上文例 9-1
    ActionListener a2 = new ActionListener(){
        public void actionPerformed(ActionEvent e){
            try{
            String factor  = (String)combo.getSelectedItem();
            Banji obj = (Banji)grade.map.get(factor);
            Vector<Student> v = obj.getV();
            String key = ((JButton)e.getSource()).getActionCommand();
            String className = grade.xmlmap.get(key);
obj.statistics((IVisitor)(Class.forName(className).newInstance()),contentPane);
            }
            catch(Exception ee){ee.printStackTrace();}
        }
    };
    public void init(String strFile){
        grade.readXMLFile("d:/info.xml");  //读访问者信息配置文件
        int y = 110, yStep = 40;
        int size = grade.xmlmap.size();
        JButton infoBtn[] = new JButton[size];
        int pos = 0;
        Set ss = grade.xmlmap.keySet();                    //获得配置文件的键集合
        Iterator<String> it = ss.iterator();
        while(it.hasNext()){
            String name = it.next();
            infoBtn[pos] = new JButton(name);          //对每一个键动态产生一个按钮
            infoBtn[pos].setBounds(10, y, 150, 30);
            infoBtn[pos].addActionListener(a2);        //加消息响应
            add(infoBtn[pos]); pos ++; y+=yStep;
        }
        //其余所有代码同上文例 9-1，但要把原代码中关于“统计功能”的界面代码去掉
    }
}
```

由于可以动态配置访问者信息文件，因此界面一定是动态生成的。本示例中是把配置文件中的键值显示在按钮上的。也就是说，若配置文件中有 n 个键值，就要动态产生 n 个按钮。当单击某一个按钮时，就要动态调用相应的具体访问者类，完成相应功能的执行。

动态生成界面关键思路：获取配置文件的键集合，根据该集合动态产生按钮集合，按钮上的文本信息就是每个键的字符串值，而且每个按钮对应相同的消息响应映射。

着重理解 actionPerformed()方法，它利用反射技术实现了具体访问者的动态调用。若想利用反射技术，必须得知道反射的字符串类名。本示例是通过以下步骤获取待反射的类名的：①获取 JButton 按钮的命令字符串，该串即是配置文件的键值；②根据该字符串反查配置文件的 map 映射，获取对应的值，该值即是具体访问者的字符串类名。

【例 9-2】 用户登录加密功能。

用户登录是 Web 程序中的重要功能，相关操作包括用户注册、用户登录检查。经常有这样的情况：初始时用户信息（例如用户名、密码）是透明的，但将来有可能对这些信息进行加密保存，那么如何预留加密接口呢？访问者模式是较好的实现方式之一。

假设仍以第 4 章生成器模式 login 表为基础，见表 4-1 说明。具体代码如下。

1. 访问者模式基本代码

```
//定义抽象访问接口 IVisitor
public interface IVisitor {
    public void visit(User u);  //表明要对用户对象进行加密
}
//初始时默认具体加密访问者 EncryptVisitor
public class EncryptVisitor implements IVisitor {
    public void visit(User u) {
    }
}
```

visit()是空方法，表明没有对用户对象进行加密，说明初始时用户名、对象是透明的。

```
//用户基础类 User
public class User {
    String user;
    String pwd;
    int type;
    public User(String user,String pwd, int type){
        this.user=user; this.pwd=pwd; this.type=type;
    }
    public User(String user, String pwd){
        this.user=user; this.pwd=pwd;
    }
    public void encrypt(IVisitor v){
        v.visit(this);
    }
}
```

encrypt()是用户对象加密方法。

2. 访问者模式功能调用代码

访问者模式框架代码加在哪些功能上呢？主要有三方面：①注册新用户要预留加密代码；②用户登录检查要预留加密代码；③整体用户信息修改要预留加密代码。这是由于初始时用户名、密码没有加密；当采用加密代码后，必须对数据库已有用户进行加密处理。因此主要编制了以下三个 servlet。

（1）注册新用户 servlet 类 Regist，url 为 regist。

```
public class Regist extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public Regist() {}
    protected void service(HttpServletRequest req, HttpServletResponse rep)
throws ServletException, IOException {
    String user = req.getParameter("user");
    String pwd = req.getParameter("pwd");
    String type = req.getParameter("type");

    IVisitor v = new EncryptVisitor();
    User u = new User(user, pwd,Integer.parseInt(type));
    u.encrypt(v);   //将用户对象 u 进行加密

    String result = "register success";
    try{
```

```
String strSQL = "select * from login where user='" +u.user+ "'";
DbProc dbobj = new DbProc();
Connection conn = dbobj.connect();
Statement stm = conn.createStatement();
ResultSet rst = stm.executeQuery(strSQL);
if(rst.next())
    result = "register failare";
else{
    strSQL = "insert into login values('" +u.user+ "'," +
                "'" +u.pwd+ "'," +u.type+ ")";
    stm.executeUpdate(strSQL);
}
            rst.close(); stm.close(); conn.close();
        }
        catch(Exception e){e.printStackTrace();}
        rep.getWriter().print(result);
    }
}
```

加密接口方法体现在 u.encrypt(v)语句上。本示例为了简单，没有编制注册输入页面，但也可方便验证该类的正确性，只需在 IE 地址栏上输入以下地址即可：http://IP:PORT/WEB 工程名/regist?user=1000&pwd=12&type=1。

（2）登录检查 servlet 类 Login, URL 为 login。

```
public class Login extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public Login() {}
    protected void service(HttpServletRequest req, HttpServletResponse rep)
throws ServletException, IOException {
        String user = req.getParameter("user");
        String pwd = req.getParameter("pwd");
        IVisitor v = new EncryptVisitor();
        User u = new User(user, pwd);
        u.encrypt(v);     //加密接口方法

        String result = "login success";
        try{
            String strSQL = "select * from login where user='" +u.user+ "'"+
                            " and pwd='" +u.pwd+ "'";
            DbProc dbobj = new DbProc();
            Connection conn = dbobj.connect();
            Statement stm = conn.createStatement();
            ResultSet rst = stm.executeQuery(strSQL);
            if(!rst.next())
                result = "login failare";
            rst.close(); stm.close(); conn.close();
        }
        catch(Exception e){e.printStackTrace();}
        rep.getWriter().print(result);
    }
}
```

加密接口方法体现在 u.encrypt(v)语句上。本示例为了简单，没有编制登录输入页面，但也可方便验证该类的正确性，只需在 IE 地址栏上输入形如以下地址即可：http://IP:PORT/WEB 工程名/login?user=1000&pwd=12。

（3）全部用户信息加密处理 servlet 类 ChangeAllUser, URL 为 changealluser。

```
public class ChangeAllUser extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public ChangeAllUser() {}
    protected void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        try{
            IVisitor v = new EncryptVisitor();
            DbProc dbobj = new DbProc();
            Connection conn = dbobj.connect();
            Statement stm = conn.createStatement();
            ResultSet rst = stm.executeQuery("select * from login");
            Vector  vec = new Vector();
            while(rst.next()){
                User u=new User(rst.getString("user"),rst.getString("pwd"),
                        rst.getInt("type"));
                vec.add(u);
            }
            rst.close();
            stm.executeUpdate("delete from login");
            for(int i=0; i<vec.size(); i++){
                User u = (User)vec.get(i);
                u.encrypt(v);    //加密用户接口方法
                String strSQL = "insert into login values('"+u.user+"',"+
                        "'"+u.pwd+"'," +u.type+ ")";
                stm.executeUpdate(strSQL);
            }
            stm.close(); conn.close();
        }
        catch(Exception e){e.printStackTrace();}
    }
}
```

由于是对已有全部用户进行加密处理，因此必须首先查询 login 表，形成用户信息 Vector 向量集合 vec；然后删除 login 表所有记录；最后遍历 vec 向量，对每个用户对象 u 利用 u.encrypt(v)进行加密。本示例为了简单，没有编制启动页面，但也可方便验证该类的正确性，只需在 IE 地址栏上输入形如以下地址即可：http://IP:PORT/WEB 工程名/changealluser。

3. 加密具体访问者类 EncryptVisitor

```
public class EncryptVisitor implements IVisitor {
    private String encode(String str) {
        String result = null;
        try{
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(str.getBytes());
            byte buf[] = md.digest();
            result = byteToString(buf);
        }
        catch (NoSuchAlgorithmException e) {}
        return result;
    }
    private String byteToString(byte[] aa) {
        String hash = "";
        for (int i = 0; i < aa.length; i++) {
            int temp;
```

```
            temp=aa[i]<0?aa[i]+256:aa[i];
            if (temp < 16)
                hash += "0";
            hash += Integer.toString(temp, 16);
        }
        hash = hash.toUpperCase();
        return hash;
    }
    public void visit(User u) {
        u.user = encode(u.user);u.pwd = encode(u.pwd);  //对用户名、密码进行加密
    }
```

本示例采用了 MD5 加密算法，MD5 全称为 Message-digest Algorithm 5（信息摘要算法），是一种不可逆的加密算法。它主要应用于确保信息传输的完整一致性与系统登录认证这两个方面。JDK 中可直接应用 java.security.MessageDigest 类对字符串进行 MD5 加密，结果是一个 128 字节缓冲区 buf，这一具体过程见示例中 encode()方法。byteToString()方法是将 buf 字节缓冲区以每 4 字节为单位，依次转化为 16 进制大写字符。由于字节缓冲区大小是 128 字节，所以最后的加密结果字符串长度为 128/4=32。

第10章 命令模式

10.1 问题的提出

顾名思义，命令模式一定是有命令发送者、命令接收者。命令发送者负责发送命令，命令接收者负责接收命令并完成具体的工作。例如，老师通知学生打扫卫生，老师是命令发送者，学生是命令接收者；学生接到命令后，完成分担区的清扫工作。再如完成科研项目过程中，负责人要求某月某日之前必须完成某部分工作，负责人相当于命令发送者，项目参加者相当于命令接收者，当接到命令后，必须按规定日期完成所属工作。毫无疑问，“命令”形式在生活中是普遍存在的，那么在计算机中如何描述呢？命令模式为我们提出了较好的设计思路。

10.2 命令模式

命令模式主要针对需要执行的任务或用户提出的请求进行封装与抽象。抽象的命令接口描述了任务或请求的共同特征，而实现则交由不同的具体命令对象完成。每个命令对象都是独立的，它负责完成需要执行的任务，却并不关心是谁调用它。

考虑老师通知学生打扫卫生的程序描述，具体代码如下。

1. 抽象命令接口 ICommand

```
interface ICommand{
    public void sweep();
}
```

2. 命令接收者 Student

```
class Student{
    public void sweeping(){
        System.out.println("we are sweeping the floor");
    }
}
```

在命令模式中，具体工作一定是在接收者中完成的，这一点非常重要。示例中“清扫”工作是由 sweeping()方法完成的。

3. 命令发送者 Teacher

```
class Teacher implements ICommand{
    private Student receiver = null;
    public Teacher(Student receiver){
        this.receiver = receiver;
```

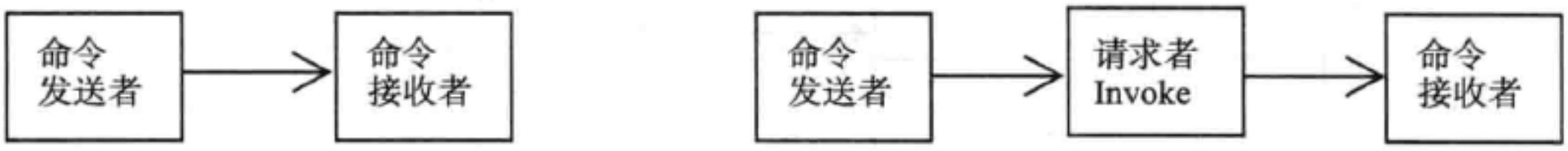
```
    }
    public void sweep(){   //发送 sweep 清扫命令
        receiver.sweeping();
    }
}
```

命令发送者类中，一般来说包含命令接收者的引用，表明发送命令的目的地址。所以 Teacher 类中定义了接收者 Student 类对象的引用。而实现的抽象接口方法中表明发送命令的具体过程，sweep()中利用方法转发说明具体的清扫工作是由接收者 Student 对象完成的。

4. 命令请求者类 Invoke

```
class Invoke{
    ICommand command;
    public Invoke(ICommand command){
        this.command = command;
    }
    public void execute(){
        command.sweep();  //启动命令
    }
}
```

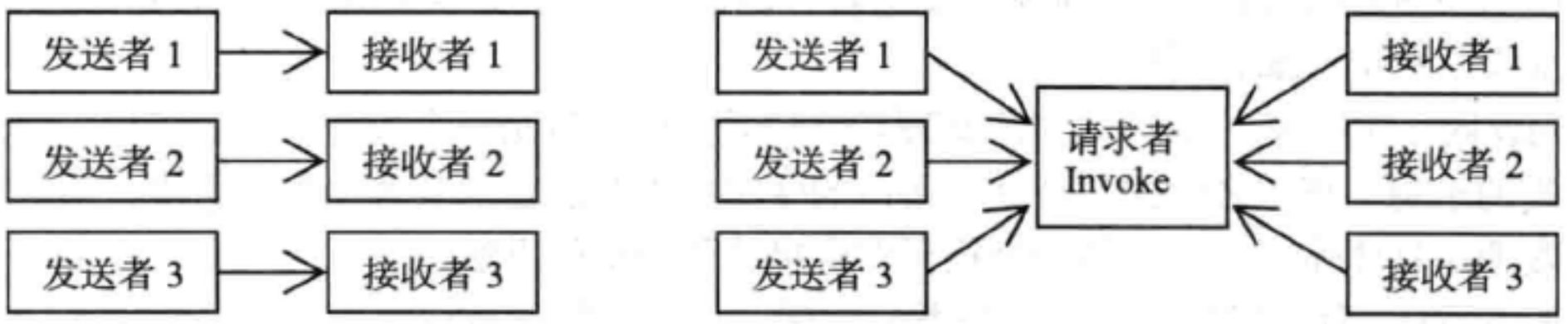
按普通思路来说，有命令发送者类、命令接收者类已经足够了，也易于理解。为什么还要有请求者类 Invoke 呢？本示例比较简单，但从简单中却能理解较深刻的道理，如下所示。



（a）普通思路　　（b）命令模式思路

图 10-1　普通思路与命令模式思路对比

普通思路是命令发送者直接作用命令接收者，而命令模式思路是在两者之间增加一个请求者类，命令发送者与请求者作用，请求者再与命令接收者作用，请求者起到了一个桥梁的作用。再看图 10-2，进一步加深对 Invoke 的感性认识。



（a）多发送者、多接收者普通思路　　（b）多发送者、多接收者命令模式思路

图 10-2　多发送者与多接收者对比

可以看出，在命令模式中，请求者是命令的管理类。还可以画出许多类似图 10-2（b）应用的功能框图。很明显，与图 10-2（a）对比，有了 Invoke，可以使层次结构更清晰，方便发送者、接收者之间的命令管理与维护。

5. 一个简单的测试类

```
public class Test {
    public static void main(String[] args) {
```

```
        Student s = new Student();          //定义接收者
        Teacher t = new Teacher(s);         //定义命令发送者
        Invoke invoke = new Invoke(t);      //将命令请求加到请求者对象中
        invoke.execute();                   //由请求者发送命令
    }
}
```

上述 main()方法功能与下述代码是一致的，并没有用到请求者 Invoke 类，这是由于本示例过于简单的缘故。请读者在学习本章后续知识的时候，一定要多思考 Invoke 的作用。

```
Student s = new Student();      //定义接收者
Teacher t = new Teacher(s);     //定义命令发送者
t.sweep();
```

通过上述，可以得出命令模式更一般的抽象 UML 类图，如图 10-3 所示。

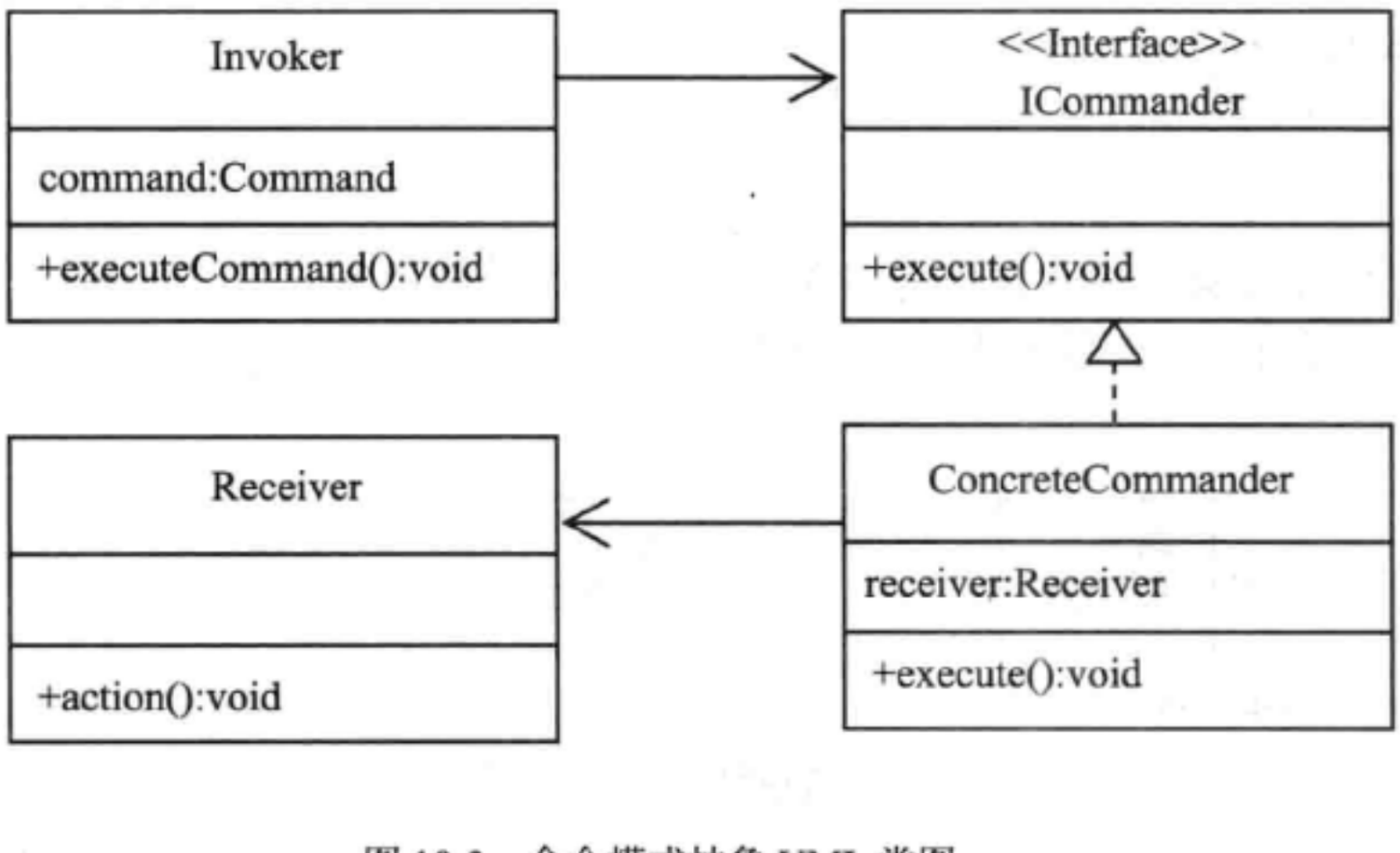


图 10-3　命令模式抽象 UML 类图

命令模式一般有 4 种角色，如下所示。

- ICommander：抽象命令者，是一个接口，规定了用来封装请求的若干个方法。
- ConcreteCommander：具体命令发送者，即命令源。它是实现命令接口的类的示例，如上文中的 Teacher 类。
- Invoker：请求者，具体命令的管理与维护类。请求者是一个包含“命令接口”变量的类的示例。请求者中的“命令”接口的变量可以存放任何具体命令的引用，请求者负责调用具体命令，让具体命令执行那些封装了请求的方法。
- Receiver：命令接收者，是一个类的示例。该示例负责执行与请求相关的操作，如上文中的 Student 类。

10.3　深入理解命令模式

10.3.1　命令集管理

考虑求任意多边形面积问题，要求按逆时针方向输入 n 个点的坐标。多边形面积的计算

公式如下：设有 n 个点（x[0],y[0]），（x[1],y[1]），...，（x[n-1],y[n-1]）围成一个没有边相交的多边形，则其围成的闭合多边形面积为 S=（∑y[i] *（x[i-1]-x[i+1]））/2，其中，i=0,1,...,n-1。若 i-1<0，则 x[i-1]=x[n-1]；若 i+1=n，则 x[i+1]=x[0]。该公式用于凸凹多边形均可，具体代码如下。

1. 计算面积类 PolyonCalc

即是命令接收响应者 Receiver。

```
class Point{
    float x,y;
    public Point(float x, float y){
        this.x=x; this.y=y;
    }
}
class PolyonCalc{
    public float getArea(Point pt[]){
        float s;
        int size = pt.length;
        if (pt.length<3) return 0;
        s=pt[0].y*(pt[size-1].x-pt[1].x);
        for (int i=1;i<size;i++)
            s+=pt[i].y*(pt[(i-1)].x-pt[(i+1)%size].x);
        return (float)s/2;
    }
}
```

2. 抽象命令发送者 ICommander

```
interface ICommander{
    float calc();
}
```

3. 具体面积命令发送者 AreaCommander

```
class AreaCommander implements ICommander{
    PolyonCalc calc;
    Point pt[];
    public AreaCommander(Point pt[], PolyonCalc calc){
        this.pt = pt; this.calc = calc;
    }
    public float calc(){
        return calc.getArea(pt);
    }
}
```

成员变量 calc 是命令接收者 PolyonCalc 对象的引用，成员变量 pt[]数组用于接收 n 个逆时针点的坐标。

4. 命令管理者 CommanderManage

相当于命令请求者 Invoke，具体代码如下。

```
class CommanderManage{
    ArrayList<ICommander> list = new ArrayList();
    public void add(ICommander c){
        list.add(c);
    }
    public void executeCommand(){
```

```
            for(int i=0; i<list.size(); i++){
                float value = list.get(i).calc();
                System.out.println("The area is:" +value);
            }
        }
    }
```

可以看出，定义了 ICommand 命令的集合成员变量 list，定义了添加命令消息的 add()方法以及整体执行命令集的 executeCommand()方法。读者可以根据需要自行丰富该命令管理类的功能。

5. 一个简单的测试类

```
public class Test {
    public static void main(String[] args) {
        CommanderManage manage = new CommanderManage();          //定义命令管理者
        PolyonCalc calc = new PolyonCalc();                      //定义命令接收者
        Point pt[] = new Point[3];
        //定义第 1 个三角形坐标
        pt[0]=new Point(0,0);
        pt[1]=new Point(1,0);
        pt[2]=new Point(0,1);
        AreaCommander com = new AreaCommander(pt, calc);         //定义第 1 条面积命令
        manage.add(com);                                         //加入命令管理者
        pt = new Point[3];
        //定义第 2 个三角形坐标
        pt[0]=new Point(0,0);pt[1]=new Point(2,0);pt[2]=new Point(0,2);
        //定义第 2 条面积命令
        AreaCommander com2 = new AreaCommander(pt, calc);
        manage.add(com2);                                        //加入命令管理者
        manage.executeCommand();                                 //统一执行
    }
}
```

该测试类主要思路：将两条计算三角形面积的命令 com、com2 加入到命令管理者 manage 中，最后调用 executeCommand()方法统一执行这两条命令对应的执行方法。

10.3.2 加深命令接口定义的理解

可能有许多读者认为命令接口很好定义，其实不然。还是用 10.3.1 节示例加以说明，假设现在要增加求多边形周长的功能，该如何实现呢？

方法 1：

由于要求多变形的面积和周长，共有两个功能，因此命令接口也要定义两个方法，如下所示。

```
interface ICommander{
    float calcArea();       //求面积
    float calcLen();        //求周长
}
```

毫无疑问，命令接口内容变了，许多已经实现的代码都要进行改变。而且，将来若需求分析发生变化，例如若计算多边形重心坐标，按照此思路，还需要修改命令接口，因此这种

命令接口思考方式仅仅处于底层状态，是不可取的，相关的具体代码也就不列举了。好的思路是下述的方法 2。

方法 2：

```
//定义抽象命令接口 ICommander
interface ICommander{
    float calc();
}
```

可以看出，虽然要实现求面积和周长两个功能，但定义的命令接口内容不变。那么，它与方法 1 中的定义有什么不同呢？从浅层角度讲，方法 1 是把“计算面积”、“计算周长”、“计算 XXX”等作为接口内容的，所以接口方法一定是多个。方法 2 是把“计算”作为接口内容，而不管“XXX”是什么，所以仅定义一个接口方法即可。从深层角度讲，定义的抽象命令接口内容不应仅适用于计算多边形的面积、周长运算，还应适用于复数、微积分等尽可能多的计算功能。因此命令接口是更高层次的抽象，读者一定要细心把握。

```
//定义多边形两种命令类
class AreaCommander implements ICommander{/*同 10.3.1 小节*/ } //面积命令类
class LenCommander implements ICommander { // 周长命令类
    PolyonCalc calc;
    Point pt[];
    public LenCommander(Point pt[], PolyonCalc calc) {
        this.pt = pt;
        this.calc = calc;
    }
    public float calc() {
        return calc.getLength(pt);
    }
}
```

LenCommander 是新增加的周长命令类。由此可知，若再增加求多边形重心坐标功能，只需定义一个重心命令类实现 ICommander 接口即可，已有的命令实现类无需改变。

```
//定义命令管理类
class CommanderManage{/*同 10.3.1 小节*/}

//定义多边形功能类 PolyonCalc
class Point{/*同 10.3.1 小节*/}
class PolyonCalc{
    public float getArea(Point pt[]){ /*同 10.3.1 小节*/}
    public float getLength(Point pt[]){  //求多边形周长
        int i=0;
        float len=0;
        for(i=0; i<pt.length-1; i++){
            len += distance(pt[i],pt[i+1]);
        }
        len += distance(pt[0],pt[pt.length-1]);
        return len;
    }
    public float distance(Point one,Point two){
        return (float)Math.sqrt((one.x-two.x)*(one.x-two.x)+(one.y-two.y)*(one.y-
two.y));
```

```
        }
    }
    //一个简单的测试类
    public class Test {
        public static void main(String[] args) {
            CommanderManage manage = new CommanderManage();
            Point pt[] = new Point[3];
            pt[0] = new Point(0, 0);
            pt[1] = new Point(1, 0);
            pt[2] = new Point(0, 1);.
            PolyonCalc calc = new PolyonCalc();
            AreaCommander com = new AreaCommander(pt, calc);     //面积命令
            manage.add(com);
            LenCommander com2 = new LenCommander(pt, calc);      //周长命令
            manage.add(com2);
            manage.executeCommand();
        }
    }
```

该测试类主要思路：将一条面积命令 com、周长命令 com2 添加到命令管理对象 manage 中。与 10.3.1 节中的测试类对比，可知 manage 可以添加不同类型的 ICommand 命令。虽然命令种类不同，但在命令管理类 executeCommand()方法中，以相同的调用形式，一个实现了求面积功能，另一个实现了求周长功能。这也再次说明最初命令接口内容定义的重要性。

10.3.3 命令模式与 JDK 事件处理

JDK 事件处理机制如图 10-4 所示。

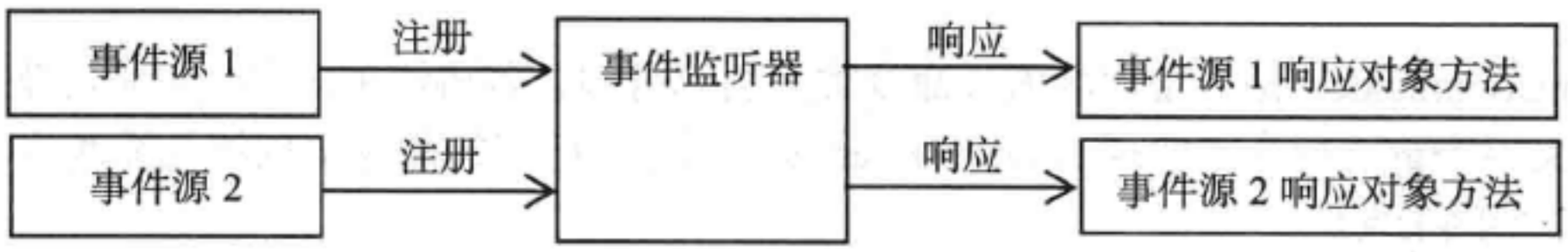


图 10-4 JDK 事件处理机制

可以看出，事件源与对应的响应方法由事件监听器割裂开来，只有注册的事件源，才有可能进行方法响应。其主要功能如下所示。

- 事件监听器是保存侦听哪些数据源的容器，既然能注册事件源，就能撤销事件源。
- 当有事件（消息）发生的时候，事件侦听器遍历监听的数据源容器，判断到底是哪个数据源产生的事件；若找到，则执行相应的消息响应方法。
- 事件监听器是 Java 消息处理机制的核心，由 Java 系统完成。因此，我们所做的只是完成注册事件源，以及注意在事件监听器机制下在程序何位置处编制相应的消息响应方法。

很明显，JDK 事件处理机制与命令模式是吻合的。事件源相当于命令发送源，事件源响应对象相当于命令接收源，事件监听器相当于命令请求者 Invoke。

考虑布局管理器演示示例，界面如图 10-5 所示。当单击“FlowLayout 演示”按钮时，界面如图 10-5（a）所示；当单击“BorderLayout 演示”按钮时，界面如图 10-5（b）所示。

（a）FlowLayout 演示界面

（b）BorderLayout 演示界面

图 10-5 布局管理器演示

用两种方法实现图示功能，具体代码如下。

方法 1：

```
public class MyFrame extends JFrame{
    JPanel content = new JPanel();
    ActionListener a = new ActionListener(){                    //命令接收者对象
        public void actionPerformed(ActionEvent e){
            content.removeAll();                                //清除所有子窗口
            content.setLayout(new FlowLayout());                //设置流式布局管理器
            content.add(new JButton("1")); content.add(new JButton("2"));
            content.add(new JButton("3")); content.add(new JButton("4"));
            content.add(new JButton("5"));
            content.updateUI();
        }
    };
    ActionListener a2 = new ActionListener(){                   //命令接收者对象
        public void actionPerformed(ActionEvent e){
            content.removeAll();                                //清除所有子窗口
            content.setLayout(new BorderLayout()); //设置方位布局管理器
            content.add(new JButton("北"),BorderLayout.NORTH);
            content.add(new JButton("南"),BorderLayout.SOUTH);
            content.add(new JButton("西"),BorderLayout.WEST);
            content.add(new JButton("东"),BorderLayout.EAST);
            content.add(new JButton("中"),BorderLayout.CENTER);
            content.updateUI();
        }
    };
    public void init(){
        JPanel p = new JPanel();
        JButton btn = new JButton("FlowLayout 演示");          //定义事件源（命令源）对象
        JButton btn2= new JButton("BorderLayout 演示");        //定义事件源（命令源）对象
        p.add(btn); p.add(btn2);

        add(p, BorderLayout.NORTH);
        add(content);

        btn.addActionListener(a);
        btn2.addActionListener(a2);
```

```
        setSize(300,150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    Public static void main(String[] args) {
        new MyFrame().init();
    }
}
```

本示例中，有效事件源（命令源）是 init()方法中的两个 JButton 对象 btn、btn2，命令接收者是类中定义的两个 ActionListener 内部匿名类对象 a、a2，那么哪个是命令请求者 Invoke 呢？命令请求者是事件监听器，由 JDK 自身隐藏，无需编制。

可以得出，该方法的特点是命令源直接用系统类，如 JButton 等，而且命令源（btn、btn2）与命令接收者（a、a2）是分割开来的。这与下述的方法 2 稍有不同。

方法 2：

```
interface IReceiver{   //抽象命令接收者方法定义，此接口方法 1 不需要
    public void response();
}
class MyFrame extends JFrame implements ActionListener{
    JPanel content = new JPanel();
    class FlowBtn extends JButton implements IReceiver{ //FlowBtn 既是命令发送者，又是
                                                         //接收者
        public FlowBtn(String name){
            super(name);
        }
        public void response() {
            content.removeAll();
            content.setLayout(new FlowLayout());
            content.add(new JButton("1")); content.add(new JButton("2"));
            content.add(new JButton("3")); content.add(new JButton("4"));
            content.add(new JButton("5"));
            content.updateUI();
        }
    }
    class BorderBtn extends JButton implements IReceiver{//BorderBtn 既是命令发送者，
                                                         //又是接收者
        public BorderBtn(String name){
            super(name);
        }
        public void response() {
            content.removeAll();
            content.setLayout(new BorderLayout());
            content.add(new JButton("北"),BorderLayout.NORTH);
            content.add(new JButton("南"),BorderLayout.SOUTH);
            content.add(new JButton("西"),BorderLayout.WEST);
            content.add(new JButton("东"),BorderLayout.EAST);
            content.add(new JButton("中"),BorderLayout.CENTER);
            content.updateUI();
        }
    }
    public void init(){
```

```
        JPanel p = new JPanel();
        FlowBtn btn = new FlowBtn("FlowLayout 演示");
        BorderBtn btn2= new BorderBtn("BorderLayout 演示");
        p.add(btn); p.add(btn2);

        add(p, BorderLayout.NORTH);
        add(content);

        btn.addActionListener(this);
        btn2.addActionListener(this);
        setSize(300,150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        IReceiver obj = (IReceiver)e.getSource();
        obj.response();
    }
    public static void main(String[] args) {
        new MyFrame().init();
    }
}
```

与方法 1 相比，主要有以下两点需要加深理解。

- 本示例中有效命令源是 JButton 的派生类 FlowBtn、BorderBtn 类型的对象。
- FlowBtn、BorderBtn 两个类既从 JButton 派生，又实现了抽象接收者接口 IReceiver，就可以看出这两个类的对象既是命令发送者，又是命令接收者。对于这一点，读者要尤为加以思考。
- FlowBtn、BorderBtn 按钮对象事件对应同一个方法 actionPerformed()，那么如何获得具体的命令接收者对象引用呢？巧妙地通过“IReceive obj=(IReceive)e.getSource()”即可完成相应功能，避免了不必要的 if 或 switch 语句。

10.3.4 命令模式与多线程

考虑这样应用，功能是连续接收字符串信息，其格式为“姓名：信息”。例如，“zhang:Hello”表明该信息是 zhang 发送的，信息内容是 hello。假设 zhang 发送的信息非常重要，因此，当接收到此人的信息时，一方面要把该信息显示在屏幕上，另一方面把该信息以 Email 形式传送给相关人士，仿真这一过程。

把上述应用划分成三部分：信息接收功能、命令监测功能、特殊信息处理功能。下面分别加以说明。

1. 信息接收功能（命令发送者）

```
interface ISource{
    boolean isbFire();
    void setbFire(boolean bFire);
}
class Msgsrc implements ISource{
    String msg;
    boolean bFire;    //重要信息标识变量
    public String getMsg(){
```

```
        return msg;
    }
    public boolean isbFire() {
        return bFire;
    }
    public void setbFire(boolean bFire) {
        this.bFire = bFire;
    }
    public void come(String msg){
        this.msg = msg;
        if(msg.startsWith("zhang:"))    //如果是 zhang 传来的信息，非常重要
            bFire = true;               //则置重要信息标识变量为 true
        //其他功能代码
    }
}
```

come()方法用于接收信息字符串，当判断是 zhang 传来的信息时，仅置成员变量 bFire 为 true，表明现在传来的字符串是重要信息。但对于重要信息有什么特殊的处理，该类根本不关心。因此，对于本类而言，相当于简化了问题的规模，易于编制。

通过 come()方法可以看出，在命令模式中，该类相当于事件源，本质上相当于命令发送者。

Msgsrc 类是从接口 ISource 派生的，这样易于扩展。ISource 接口中仅定义了设置、获得子类 bFire 成员变量的方法，这是因为在命令监测功能中要用到这两个方法。

2. 特殊信息处理功能(命令接收者)

```
interface IReceiver{
    void process(ISource src);
}
class ShowReceive implements IReceiver{
    public void process(ISource src){
        Msgsrc obj = (Msgsrc)src;
        System.out.println(obj.getMsg());
    }
}
class EMailReceive implements IReceiver{
    public void process(ISource src){
        System.out.println("this is EMail process");
    }
}
```

按照题目要求，对特殊信息要进行屏幕显示及 Email 发送处理，因此要定义两个具体接收者。

3. 命令监测功能（相当于 Invoker）

```
//定义抽象命令接口 ICommand
interface ICommand{
    public void noticeCommand();
}

//定义具体命令类
class Command implements ICommand{
    IReceiver rvr;
    ISource  src;
    public Command(IReceiver rvr, ISource src){
```

```
            this.rvr = rvr;
            this.src = src;
        }
        public void noticeCommand(){
            rvr.process(src);
        }
    }
```

本类是单一的命令类。由于示例有两个具体的接收者 ShowReceive、EMailReceive，有两种命令需要管理，因此定义了命令集管理类 CommandManage，代码如下。

```
//定义命令集管理类 CommandManage
class CommandManage extends Thread{
    Vector<ICommand> v = new Vector();
    ISource src ;
    boolean bMark = true;
    public CommandManage(ISource src){
        this.src = src;
    }
    public void addCommand(ICommand c){
        v.add(c);
    }
    public void run(){
        while(bMark){
            if(!src.isbFire())    //若非重要消息，返回
                continue;
            for(int i=0; i<v.size(); i++){//遍历命令集和，发送通知消息
                v.get(i).noticeCommand();
            }
            src.setbFire(false);//重要消息标识位复位
        }
    }
}
```

根据题目要求，需要一边传送字符串信息，同时进行命令源监测。因此 CommandManage 命令监测类是用多线程完成的。

CommandManage 类定义了成员变量 src，表明要对 ISource 类型的对象状态进行监测。成员变量 v 是 ICommand 类型的对象的集合。

run()方法表明了命令源 src 监测的具体过程。首先利用 isFire()方法获得 src 当前状态值，当为 true 时，表明有重要信息到来了，要进行特殊处理；然后遍历命令集合，发送通知命令进行响应；最后复位命令源标识位，等待下一次重要消息的到来。

4. 一个测试类

```
public class Test{
    public static void main(String args[])throws Exception  {
        Msgsrc src = new Msgsrc();                          //定义事件源（发送者）
        ShowReceive rvr = new ShowReceive();                //定义接收者
        EMailReceive rvr2= new EMailReceive();              //定义接收者 2
        Command com = new Command(rvr,src);                 //定义命令 1
        Command com2= new Command(rvr2,src);                //定义命令 2

        CommandManage manage = new CommandManage(src);//定义命令集 Invoke
```

```
manage.addCommand(com);                          //命令集加入命令 1
manage.addCommand(com2);                         //命令集加入命令 2
manage.start();                                  //启动命令集监测线程

String s[]={"li:aaa","zhang:hello","li:bbb"};
for(int i=0; i<s.length; i++){
    src.come(s[i]);                              //仿真向命令源传送字符串
    Thread.sleep(1000);
}
manage.bMark = false;                            //设置结束线程标识
    }
}
```

10.4 应 用 探 究

【例 10-1】 简单记事本程序设计与实现。

功能包括新建、打开、保存、退出等，主界面如图 10-6 所示。

图 10-6 简单记事本主界面

事实上，Swing 中的菜单已经体现出了命令模式思想，事件源是菜单中的子菜单项，事件响应是由监听器完成的，而监听过程是由 JDK 实现的。

10.3.3 节示例中利用 ActionListener 接口实现了事件响应，本示例采用 Action 接口实现相应功能。Action 接口是从 ActionListener 接口派生的，JDK 提供了该接口的默认实现抽象类 AbstractAction。因此，若定义事件响应类，一般都要从该抽象类派生。为了讲解清楚，分成两部分加以说明：界面生成及事件初始化，事件响应。

1. 界面生成及事件初始化

```
class MyFrame extends JFrame{
    JTextArea ta = new JTextArea();              //文本区
    File curFile = null;                         //当前打开文件对象
    private Action[][] actions = {               //4 个响应类
        {new NewAction(),new OpenAction(),new SaveAction(),new ExitAction()}
};
    private void saveFile(){
    try{
        FileOutputStream out = new FileOutputStream(curFile);
        byte buf[] = ta.getText().getBytes();
```

```
            out.write(buf);
            out.close();
        }catch(Exception e){e.printStackTrace();}
        }
        private void savePathFile(){
            JFileChooser chooser = new JFileChooser();
            int ret = chooser.showSaveDialog(MyFrame.this);
            if (ret != JFileChooser.APPROVE_OPTION)
            return;
            try{
                curFile = chooser.getSelectedFile();
                FileOutputStream out = new FileOutputStream(curFile);
                out.write(ta.getText().getBytes());
                out.close();
            }
            catch(Exception ee){ee.printStackTrace();}
        }
        private void preProcess(){
        if(curFile != null){
            saveFile();
            return;
        }
        if(!ta.getText().equals(""))
            savePathFile();
        }
        protected JMenuBar createMenubar() {
            JMenuBar mb = new JMenuBar();                    //创建菜单条
            String[] menuKeys = {"File"};
            String[][] itemKeys = {{"New","Open","Save","Exit"}};
            for (int i = 0; i < menuKeys.length; i++) {
                JMenu m = new JMenu(menuKeys[i]);  //创建菜单
                for (int j = 0; j < itemKeys[i].length; j++) {
                    JMenuItem mi = new JMenuItem(itemKeys[i][j]);//创建菜单项
                    mi.addActionListener(actions[i][j]);//菜单项添加消息响应
                    m.add(mi);          //菜单添加菜单项
                }
                mb.add(m);              //菜单条添加菜单
            }
            return mb;
        }
        public void init(){
            this.setJMenuBar(createMenubar());      //设置菜单
            JScrollPane sp = new JScrollPane(ta);   //设置滚动文本区
            add(sp);                                //添加滚动面板
            setSize(600,500);
            setVisible(true);
        }
        public static void main(String[] args) {
                new MyFrame().init();
        }
}
```

JTextArea 类型成员变量 ta 用于存放文本，File 类型成员变量 curFile 表示当前打开的文

件对象，可以根据 curFile 获得文件路径、长度等参数。因此从这一角度来说，要比定义字符串成员变量（代表当前打开文件路径）好。Action 类型成员变量 actions 是二维数组，这与菜单特点一致。从 actions 数组第一维长度可看出共有几个菜单项，从第二维长度上可看出菜单项有多少个子项，每个子项对应的事件响应类是什么。

createMenuBar()包含了创建菜单条的全过程，建立了菜单子项与事件响应数组 actions 的关联。定义的局部变量一维数组 menuKeys 代表菜单项的名称，二维数组 itemKeys 代表每个菜单子项的名称。有了这两个数组，添加菜单条就可以用循环来实现了。因此在图形用户界面中，巧妙运用数组思维可以简化代码规模，而且层次清晰。

2. 事件响应类

由于“File”菜单有 4 个菜单子项，因此根据命令模式思想，一定有 4 个单独的实现类。这 4 个类是 MyFrame 的内部类，方便与外部类的通信。

（1）New 响应类 NewAction。

```
class NewAction extends javax.swing.AbstractAction {
        public void actionPerformed(ActionEvent e) {
            preProcess();      // 前处理
            ta.setText("");    // 设文本区域内容为空
            curFile = null;    // 当前文件对象为 null
            MyFrame.this.setTitle("无标题");// 设置标题
        }
    }
```

当选中“new”命令，执行 actionPerformed()方法后，要运行前处理 preProcess()方法。该方法完成了两部分功能：①若有当前打开文件，必须先保存该文件，但无须运行文件保存对话框，对应的方法是 saveFile()；②若无打开文件，但文本区不为空，即当前“new”命令前执行的仍是“new”命令，则必须运行文件保存对话框程序，对应的方法是 savePathFile()。

（2）Open 响应类 OpenAction。

```
class OpenAction extends javax.swing.AbstractAction {
    public void actionPerformed(ActionEvent e) {
        preProcess(); // 前处理
        JFileChooser chooser = new JFileChooser();
        int ret = chooser.showOpenDialog(MyFrame.this);
        if (ret != JFileChooser.APPROVE_OPTION)
            return;
        try {
            curFile = chooser.getSelectedFile();
            if (curFile.isFile() && curFile.canRead()) {
                int size = (int) curFile.length();
                byte[] buf = new byte[size];
                FileInputStream in = new FileInputStream(curFile);
                in.read(buf);
                in.close();

                String s = new String(buf);
                ta.setText(s);
                MyFrame.this.setTitle(curFile.getName());
            }
        } catch (Exception ee) {
            ee.printStackTrace();
```

```
            }
        }
    }
```

“Open”命令与“new”命令运行流程是相似的，相同是都要进行前处理，不同是“Open”命令要运行打开文件对话框程序，选择文件，读文件，将内容显示在界面文本区域中。

（3）Save 响应类 SaveAction。

```
class SaveAction extends javax.swing.AbstractAction {
    public void actionPerformed(ActionEvent e) {
        if (curFile != null) {     // 对已打开文件保存
            saveFile();
            return;
        }
        savePathFile();            // 对前一命令是“new”的文件保存
    }
}
```

该方法含义是若对已打开文件保存，则直接调用 saveFile()保存即可。对前一命令是“new”对应的文件保存，必须显示文件保存对话框，设置保存文件名称，这是通过调用 savePathFile()方法完成的。

（4）Exit 响应类 ExitAction。

```
class ExitAction extends javax.swing.AbstractAction {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

【例 10-2】 利用配置文件和反射技术实现例 10-1 功能。

已知配置文件 config.xml 如表 10-1 所示。

表 10-1　　配置文件 config.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="menubar">File</entry>
<entry key="Filemenu">New Open Save</entry>
<entry key="Fileaction">NewAction OpenAction SaveAction</entry>
</properties>
```

配置文件共包含三部分内容：①菜单项，对应的关键字是 menubar。该文件表示目前仅有一个菜单项 File，若有多个，中间用单空格隔开即可，如“File Edit Tool”。②菜单子项，表示每个菜单的下拉子项总内容，中间用单空格隔开，关键字由“菜单项名称+menu”组成。例如 File 菜单项对应的菜单子项关键字为“File+menu=Filemenu”。③事件响应类，表示每个菜单包含的菜单子项对应的事件响应类字符串总名称，中间用单空格隔开即可，关键字由“菜单项名称+action”组成。例如，File 菜单项对应的菜单子项事件响应类关键字为“File+action=Fileaction”。

配置文件中的内容是关联的，最关键的是 menubar 关键字。通过它可知菜单项的个数，从而可知每个菜单项有多少个子菜单，每个子菜单对应的事件响应类是什么。因此本示例的主要工作是根据配置文件内容动态加载菜单，并且根据反射机制动态加载事件响应类对象。为了讲解清晰，程序共分为三部分：回调接口、主控程序、事件响应类程序。

1. 回调接口 IFileInter

```
interface IFileInter{
    void setFile(File f);
    File getFile();
}
```

该接口在主控程序、事件响应类程序中都要用到。主控程序调用事件响应类是必然的，那么事件响应类如何回调主控程序呢？利用接口技术去实现是一个较好的方法，详细说明见事件响应类部分。

2. 主控程序 MyFrame

```
class MyFrame extends JFrame implements IFileInter{
    private JTextArea ta = new JTextArea();
    private File curFile = null;
    private String menuKeys[];          //菜单项数组
    private String itemKeys[][];        //菜单子项数组
    private String actKeys[][];         //菜单子项事件响应类字符串名称数组
    private Action[][] actions = null; //菜单子项事件响应类对象数组

    class ExitAction extends AbstractAction {
        public void actionPerformed(ActionEvent e) {System.exit(0);}
    }
    protected JMenuBar createMenubar() {  //动态添加菜单及响应事件
        JMenuBar mb = new JMenuBar();
        for (int i = 0; i < menuKeys.length; i++) {
            JMenu m = new JMenu(menuKeys[i]);
            for (int j = 0; j < itemKeys[i].length; j++) {
                JMenuItem mi = new JMenuItem(itemKeys[i][j]);
                m.add(mi);
                mi.addActionListener(actions[i][j]);
            }
            mb.add(m);
        }
        JMenu menu = mb.getMenu(0);    //添加 Exit 退出菜单命令事件
        JMenuItem item = new JMenuItem("Exit");
        item.addActionListener(new ExitAction());
        menu.add(item);
        return mb;
    }
    private void createAction(){   //动态加载事件响应类对象
    actions = new Action[actKeys.length][];
    try{
        for(int i=0; i<actKeys.length; i++){
            actions[i] = new Action[actKeys[i].length];
            for(int j=0; j<actKeys[i].length; j++){
                Class classInfo = Class.forName(actKeys[i][j]);
                        Constructor c = classInfo.getConstructor(new Class[]
                                {JFrame.class,JTextArea.class});
                actions[i][j]=(Action)c.newInstance(new Object[]{this,ta});
            }
        }
    }
    catch(Exception e){e.printStackTrace();}
```

```
}
private void initConfig(){//读配置文件
try{
    Properties p = new Properties();
    p.loadFromXML(new FileInputStream("d:/config.xml"));
    String s = p.getProperty("menubar");    //获得总菜单项内容
    menuKeys = s.split("");                 //拆分后填充具体的菜单项数组
    itemKeys = new String[menuKeys.length][];
    for(int i=0; i<menuKeys.length; i++){
        s = p.getProperty(menuKeys[0]+"menu");//获得对应的菜单子项总内容
        itemKeys[i] = s.split("");          //拆分后填充具体的菜单子项数组
    }
    actKeys = new String[menuKeys.length][];
    for(int i=0; i<menuKeys.length; i++){
        s = p.getProperty(menuKeys[0]+"action");//获得对应的菜单子项总事件响应类名
        actKeys[i] = s.split("");//拆分后填充具体的菜单子项事件响应类数组
    }
}
catch(Exception e){e.printStackTrace();}
}
public void init(){
    initConfig();     //读配置文件
    createAction();   //动态加载事件响应类
    this.setJMenuBar(createMenubar());//动态加载菜单
    JScrollPane sp = new JScrollPane(ta);
        add(sp);
        setSize(600,500);
        setVisible(true);
    }
    public void setFile(File f) {curFile = f;}
    public File getFile(){return curFile;}
    public static void main(String[] args) {
        new MyFrame().init();
    }
}
```

initConfig()完成读配置文件功能，填充了成员变量菜单项一维数组 menuKeys[]、菜单子项二维数组 itemKeys[][]、事件响应类名二维数组 actKeys[][]的内容。

createAction()主要是利用反射技术，根据 actKeys[][]数组内容，动态加载了具体的事件响应类对象，结果存放在 Action 类型的成员变量二维数组 actions[][]中。但要注意，动态加载的事件响应类构造方法是含参的，包含两个参数，从“c.newInstance(**new** Object[]{**this**,ta})”就可看出来，一个是 JFrame 对象，另一个是 JTextArea 对象。这是由于按反射技术来说，事件响应类一定是外部类，这与例 10-1 不同，因此必须把必要的参数传到事件响应类中。所以构造方法一般来说是有参的。

createMenubar()是根据 menuKeys[]、itemKeys[][]、actions[][]动态加载菜单及建立事件关联。由于 Exit 退出命令无需动态加载，因此程序中把该项加在第 1 条菜单的最后一个子项上。

可以看出，该类实现了 IFileInter 接口，setFile()、getFile()方法主要是用于事件响应类回调的。

3. 事件响应类

由于采用了配置文件及反射技术，程序的柔性更好。例如，若增加事件响应类，只需修改配置文件，主控程序无需修改。本部分仅以 NewAction 类说明，其他响应类均大同小异。

```
class NewAction extends AbstractAction {
    JFrame frm;
    JTextArea ta;
    File curFile;

    public NewAction(JFrame frm, JTextArea ta) {
        this.frm = frm;
        this.ta = ta;
    }

    private void saveFile() {
        try {
            FileOutputStream out = new FileOutputStream(curFile);
            byte buf[] = ta.getText().getBytes();
            out.write(buf);
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void savePathFile() {
        JFileChooser chooser = new JFileChooser();
        int ret = chooser.showSaveDialog(frm);
        if (ret != JFileChooser.APPROVE_OPTION)
            return;
        try {
            curFile = chooser.getSelectedFile();
            FileOutputStream out = new FileOutputStream(curFile);
            out.write(ta.getText().getBytes());
            out.close();
        } catch (Exception ee) {
            ee.printStackTrace();
        }
    }

    private void preProcess() {
        curFile = ((IFileInter) frm).getFile();      // 必须回调主控程序获得 File 变量
        if (curFile != null) {
            saveFile();
            return;
        }
        if (!ta.getText().equals(""))
            savePathFile();
    }

    public void actionPerformed(ActionEvent e) {
        preProcess();
        ta.setText("");
        curFile = null;
        ((IFileInter) frm).setFile(curFile);    // 必须回调主控程序设置 File 变量
```

```
            frm.setTitle("无标题");
        }
    }
```

该流程与例 10-1 中一致，就不多说了。这里着重分析为什么要用到 IFileInter 接口。从开发方便角度来说，成员变量 frm 是 JFrame 类型，而不是 MyFrame 类型。若是 MyFrame 类型，则程序就捆绑死了；没有 MyFrame 类，该类也就无从编码了。那么这就涉及一个问题，由于在事件响应类中执行 New、Open、Save 命令后，文件状态发生了变化，因此必须通知主控程序。而 frm 是 JFrame 类型，因此不可能直接与 MyFrame 对象通信。但是我们可以巧妙地定义接口 IFileInterface，让 MyFrame 实现该接口，这样 frm 既是 JFrame 又是 IFileInter，把 frm 强制转换成 IFileInter 引用对象就可以了。所以只需在 IFileInter 接口中定义回调所需通信方法即可。

saveFile()、savePathFile()等方法在本类中出现，也可能在其他事件响应类如 OpenAction 中出现，这是不可避免的。任何事物都是一分为二的。利用反射技术方便合作开发，也必然带来一些通信、代码重复问题。

第 11 章　装饰器模式

11.1　问题的提出

在消息日志功能中，接收到的消息可以直接送往屏幕显示，也可以用文件保存，其功能类 UML 类图如图 11-1 所示。

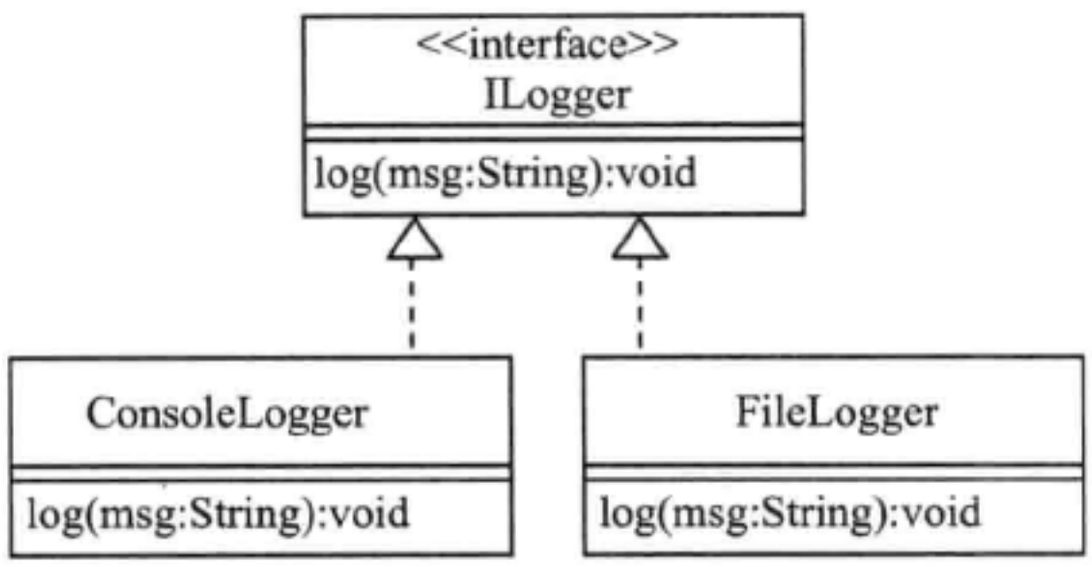


图 11-1　消息日志 UML 类图

不考虑消息日志功能的全部实现过程，仅考虑 ILogger 的实现类，具体代码如下。

```
interface ILogger{
    void log(String msg);
}
class ConsoleLogger implements ILogger{
    public void log(String msg){
        System.out.println(msg);
    }
}
class FileLogger implements ILogger{
    public void log(String msg){
        DataOutputStream dos = null;
        try {   //为了方便，日志文件定为 d:/log.txt
          dos = new DataOutputStream(new FileOutputStream("d:/log.txt",true));
          dos.writeBytes(msg+"\r\n");
          dos.close();
        } catch (Exception e) {e.printStackTrace();}
    }
}
```

假设现在需求分析提出了新要求，接收到的信息可转化成大写字母或转化成 XML 文档，然后屏幕显示或日志保存。常规思路是利用派生类实现，增加的类如表 11-1 所示。

表 11-1　　ILogger 的派生类

子类	父类	功能
UpFileLogger	FileLogger	转化成大写字母后保存到日志文件中
UpConsoleLogger	ConsoleLogger	转化成大写字母后屏幕显示
XMLFileLogger	FileLogger	转化成 XML 格式后保存到日志文件中
XMLConsoleLogger	ConsoleLogger	转化成 XML 格式后屏幕显示

如果按照继承思路，若需求分析继续变化，则类的数目增加非常快。那么，有没有更好的解决方法呢？装饰器模式是较好的思路之一。

11.2　装饰器模式

装饰器模式利用包含代替继承，动态地给一个对象添加一些额外的功能。以消息日志功能为例，其装饰器模式 UML 类图如图 11-2 所示。

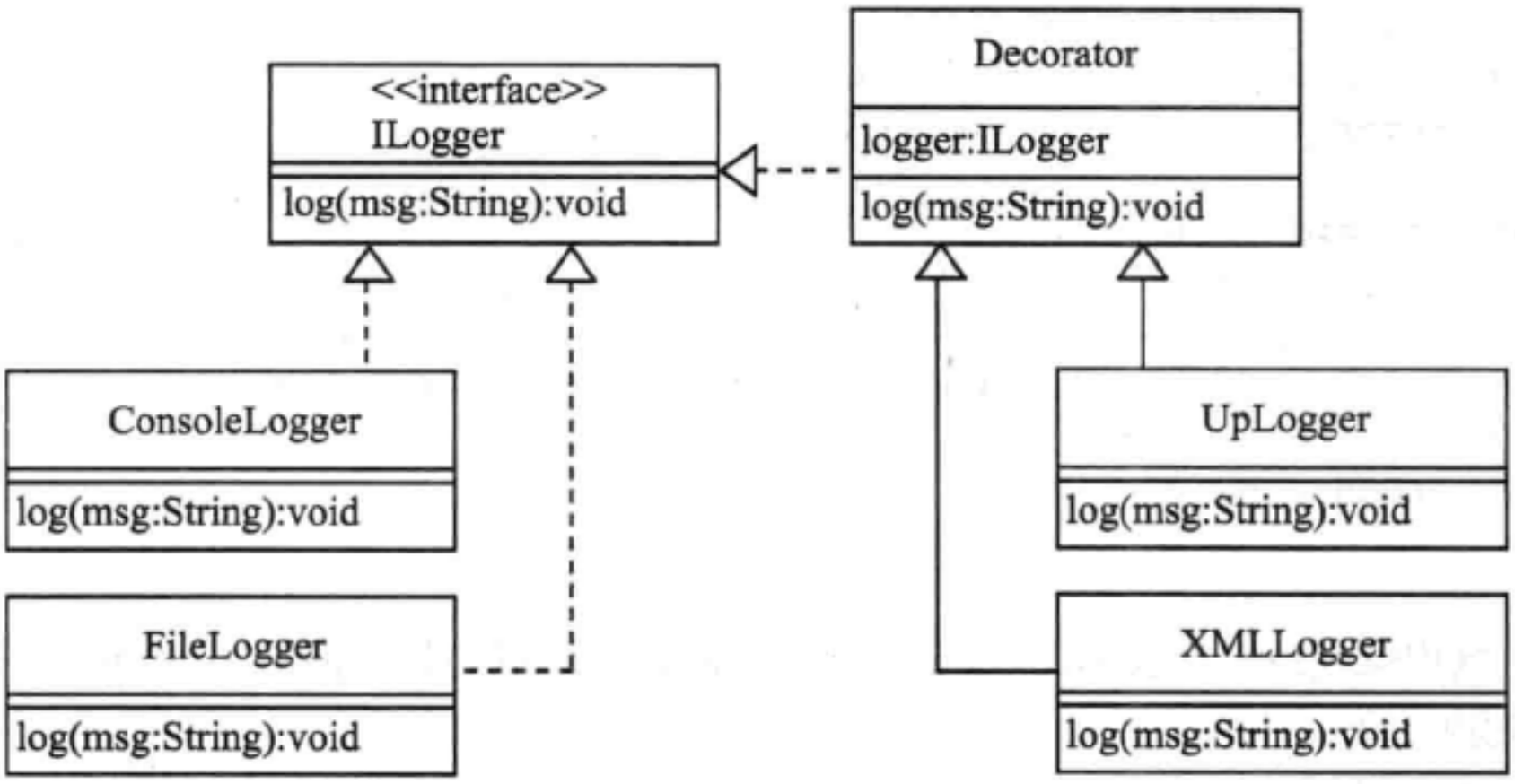


图 11-2　日志功能装饰器模式 UML 类图

可以看出，图 11-2 的左半部分与图 11-1 是相同的，右半部分是采用装饰器模式后新增的类图，具体代码如下。

1. 抽象装饰器基类 Decorator

```
abstract class Decorator implements ILogger{
    protected ILogger logger;
    public Decorator(ILogger logger){
        this.logger = logger;
    }
}
```

主要体会“implements ILogger”中的“ ILogger”与定义的成员变量“ILogger logger”中的“ILogger”语义有什么不同。需求分析变化了，但无论怎么变，它终究还是一个日志类，因此 Decorator 类要从接口 ILogger 派生。而成员变量 logger 表明 Decorator 对象要对已有的 logger 对象进行装饰，也就是说，要对已有的 FileLogger 或 ConsoleLogger 对象进行装饰。但

是由于装饰的内容不同，因此该类只能是抽象类，具体装饰内容由子类完成，如下所示。

2. 具体装饰类

```
//信息大写装饰类 UpLogger
class UpLogger extends Decorator{
    public UpLogger(ILogger logger){
        super(logger);
    }
    public void log(String msg) {
        msg = msg.toUpperCase();        //对字符串进行大写装饰
        logger.log(msg);                //再执行已有的日志功能
    }
}
```

本类中 log()方法先对字符串进行大写“装饰”，再执行已有的日志功能。若已有日志功能有 *n* 个，则装饰后的字符串可能有 *n* 个去处，也就是说，该类可以表示 *n* 个动态含义。若按 11.1 节继承模式（见表 11-1）编程，则需要编制 *n* 个具体的类，从中可知装饰器模式是采用动态编程的，缩小了程序的规模。

```
//XML 格式化装饰类 XMLLogger
class XMLLogger extends Decorator{
    public XMLLogger(ILogger logger){
        super(logger);
    }
    public void log(String msg) {
        String s = "<msg>\r\n" +
                   "<content>"+msg+"</content>\r\n"+
                   "<time>" + new Date().toString()+ "</time>\r\n"+ .
                   "</msg>\r\n";
        logger.log(s);
    }
}
```

每个信息用<msg>标签表示，<content>标签表示字符串的具体内容，<time>标签表示当前信息的采集时间。

3. 一个简单的测试类

```
public class Test {
    public static void main(String[] args)throws Exception {
        ILogger existobj = new FileLogger();      //已有的日志功能
        ILogger newobj= new XMLLogger(existobj); //新的日志装饰类，对 existobj 装饰
        String s[] = {"how","are","you"};         //仿真传送的字符串信息数组
        for(int i=0; i<s.length; i++){
            newobj.log(s[i]);
            Thread.sleep(1000);                          //每隔 1 s 传送一个新的字符串
        }
        System.out.println("End");
    }
}
```

综合图 11-2 及上述示例代码，可得装饰器模式主要有如下 4 种角色。

- 抽象构件角色（Component）：

它是一个接口，封装了将要实现的方法，如 ILogger。

• 具体构件角色（ConcreteComponent）：

它是多个类，该类实现了 Component 接口，如 FileLogger、ConsoleLogger。

• 装饰角色（Decorator）：

它是一个抽象类，该类也实现了 Component 接口，同时也必须持有接口 Component 的对象的引用，如事例中 Decorator。

• 具体的装饰角色（Decorator 类的子类，可以有一个，也可以有多个）：

这些类继承了类 Decorator，实现了 Component 接口，描述了具体的装饰过程，如 UpLogger、XMLLogger。

11.3　深入理解装饰器模式

11.3.1　具体构件角色的重要性

先考虑生活中一个实际的例子：一本菜谱书已在全国发行，特点是具有通用性，但没有考虑地域差异。假设以做白菜和大头菜为例，实际情况是以菜谱为蓝本，再考虑地域差异，比如甲地喜欢吃辣的，乙地喜欢吃甜的，用计算机如何描述呢？代码如下所示。

1. 定义抽象构件角色 ICook

```
interface ICook{
    void cook();    //做菜
}
```

2. 定义做白菜、大头菜具体角色

```
class Vegetable implements ICook{
    public void cook(){ /*按菜谱做白菜*/ }
}
class Cabbage implements ICook{
    public void cook(){ /*按菜谱做大头菜*/ }
}
```

3. 定义抽象装饰器 Decorator

```
abstract class Decorator implements ICook{
    ICook obj;    //要进一步改进菜谱
    public Decorator(ICook obj){this.obj = obj;}
}
```

4. 定义具体装饰器

```
class PepperDecorator extends Decorator{ //甲地对所有菜谱菜添加辣椒
    public PepperDecorator(ICook obj){
        super(obj);
    }
    private void addPepper(){
        //添加辣椒
    }
    public void cook(){
        addPepper();
        obj.cook();
    }
```

```
}
class SugarDecorator extends Decorator{ //乙地对所有菜添加白糖
    public SugarDecorator(ICook obj){
        super(obj);
    }
    private void addSugar(){
        //添加白糖
    }
    public void cook(){
        addSugar();
        obj.cook();
    }
}
```

Vegetable 和 Cabbage 是具体角色，也就是说，菜谱中每道菜的做法相当于具体角色，它们都是长期经验的总结。没有这些具体角色，也就谈不上与地域相关的特色装饰菜了。好的菜谱有一个重要的特点，一般来说就是只要人们能想到的菜，就能在菜谱中找到。转化成计算机专业术语即是：具体角色相当于底层的具体实现，有哪些实现功能非常重要。如果做得好的话，很大程度上，上层功能就相当于对这些底层功能进行进一步封装和完善，类似于装饰的功效。操作系统的内核是固定的，有着强大的原子功能。从广义角度来说，这些原子功能相当于具体角色。但是基于操作系统内核的应用程序是千差万别的。笔者认为，装饰模式一定起着较大的作用。

11.3.2 JDK 中的装饰模式

JDK 中应用装饰模式最广的是 IO 输入输出流部分。例如，部分字符流 UML 类图如图 11-3 所示。

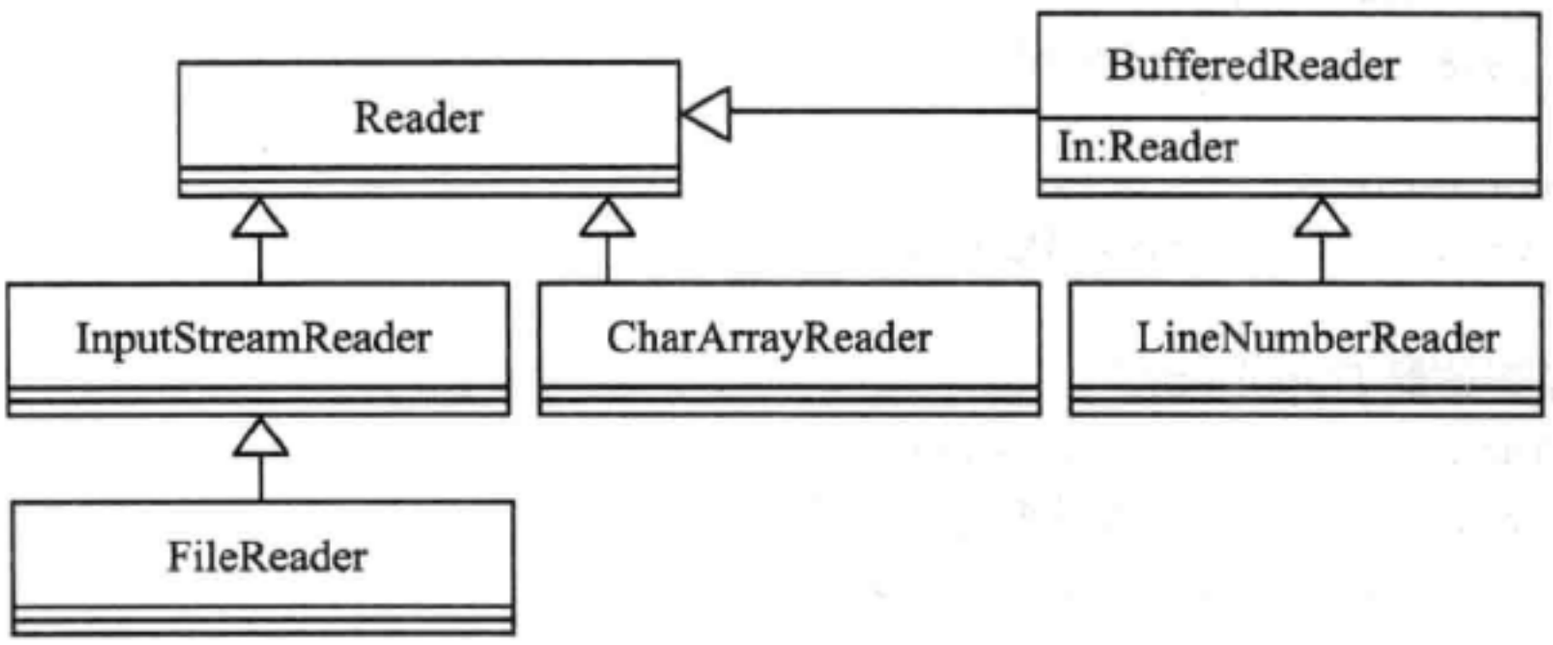


图 11-3　部分字符流 UML 类图

该类图与装饰模式类图是相似的。抽象构件相当于 Reader，具体构件相当于 InputStreamReader、CharArrayReader 以及 FileReader。该类图中并没有体现出抽象装饰器，BufferedReader、LineNumberReader 都是具体装饰器，因为它们都有 Reader 类型的成员变量 in。

那么，除了应用这些功能强大的 IO 流之外，能否自定义 IO 流而还不至于自己编制过多的代码呢？答案是可以。例如，BufferedReader 类中有 readLine()方法，本质上是按‘\r\n’进行字符串拆分，现在想按指定的字符一边读缓冲流一边进行拆分。可以修改 JDK 源码中的 BufferedReader.java，该源码在 JDK 安装目录下的 src 文件夹中。修改的具体代码如下。

```
class MyBufferedReader extends BufferedReader{

        /*
          所有成员变量从 BufferedReader.java 中完全拷贝
        */

public MyBufferedReader(Reader in, int sz) {
        super(in, sz);
        if (sz <= 0)
            throw new IllegalArgumentException("Buffer size <= 0");
        this.in = in;
        cb = new char[sz];
        nextChar = nChars = 0;
    }
private void fill() throws IOException {
        /*
          代码从 BufferedReader.java 中完全拷贝
        */
    }
    String readToken(char delim) throws IOException {
    StringBuffer s = null;
    int startChar;
    boolean omitLF = skipLF;
    bufferLoop:
        for (;;) {
        if (nextChar >= nChars)
            fill();
if (nextChar >= nChars) {
    if (s != null && s.length() > 0)
        return s.toString();
    else
        return null;
}
boolean eol = false;
char c = 0; int i;                          //下一行与源码稍有不同
if (omitLF && (cb[nextChar] == delim||cb[nextChar]=='\n'||cb[nextChar]
                    =='\r'))
    nextChar++;
skipLF = false; omitLF = false;
charLoop:
    for (i = nextChar; i < nChars; i++) {
        c = cb[i];
        if (c == delim || c=='\n' || c=='\r') {     //该行与源码稍有不同
            eol = true; break charLoop;
        }
    }
startChar = nextChar; nextChar = i;
if (eol) {
    String str;
    if (s == null) {
            str = new String(cb, startChar, i - startChar);
    } else {
        s.append(cb, startChar, i - startChar);
        str = s.toString();
    }
```

```
            nextChar++;
            if (c == delim || c=='\n' || c=='\r')  {          //该行与源码稍有不同
                skipLF = true;
            }
            return str;
        }
            if (s == null)
                s = new StringBuffer(defaultExpectedLineLength);
            s.append(cb, startChar, i - startChar);
        }
    }
```

本示例中构件缓冲流 MyBufferedReader 的步骤是：从 BufferedReader 类派生，拷贝 BufferedReader 中所有的成员变量、构造方法（仅第 1 行与源码稍有不同）及 fill()方法。最后修改 readLine()，本示例将它变成了 readToken()，仅几处代码作了修改，见注释。到此为止，我们的工作就是拷贝、粘贴加极少量的修改，没有编制一条独立的代码。但功能却强大了，可按我们指定的字符进行拆分工作。

JDK 流类中代码都比较小，与其他包中的源码相比是比较易读的，均可通过装饰模式的封装，与实际应用需求相结合，争取操作一次 IO 流，就能直接得到所需的结果。

一个问题是：BufferedReader 类中将所有的成员变量及最关键的 fill()方法都设置成私有类型，也许是更有深意。但笔者认为，将成员变量及 fill()方法设置成保护类型更好，这样示例中只需完成 readToken()即可，无须拷贝 BufferedReader 中的成员变量及 fill()方法了。

假设文本文件 data.txt 格式如表 11-2 所示。测试类功能是按“-”拆分，获得每个单词，代码如下。

表 11-2　　data.txt 文件格式

```
how-are-you
fine-thanks
well
```

```
public class Test3 {
    public static void main(String[] args)throws Exception {
        FileReader in = new FileReader("d:/data.txt");
        MyBufferedReader in2 = new MyBufferedReader(in, 1024);
        String s = "";
        while((s=in2.readToken('-')) != null){
            System.out.println(n); n ++;
            System.out.println(s);
        }
        in2.close();
        in.close();
    }
}
```

11.4 应 用 探 究

【例 11-1】 目录拷贝程序的设计与实现（主界面如图 11-4 所示）。

图 11-4　目录拷贝程序主界面

含义是将源目录中的内容（包含子目录）复制到目标目录下。

复制命令的原子功能是单文件复制，因此把该功能作为具体构件功能；目录复制是文件复制的集合，因此把该功能作为具体装饰构件功能。具体代码如下。

1. 定义抽象构件接口 ICopy

```
interface ICopy{
    void copy(String src,String dest)throws Exception;
}
```

单文件复制时，src、dest 分别表示源文件和目标文件的绝对路径；目录拷贝时，src、dest 分别表示源文件夹和目标文件夹的绝对路径。

2. 定义单文件拷贝具体构件 Copy

```
class Copy implements ICopy{
    public void copy(String srcFile, String destFile)throws Exception{
        File file = new File(srcFile);
        FileInputStream in = new FileInputStream(srcFile);
        FileOutputStream out = new FileOutputStream(destFile);
        byte buf[] = new byte[(int)file.length()];
        in.read(buf); out.write(buf);
        in.close(); out.close();
    }
}
```

复制算法比较简单。先计算源文件的长度，开与之匹配的缓冲区 buf，读源文件内容至 buf，再将 buf 写入目标文件中。

3. 定义抽象复制装饰构件 Decorator

```
abstract class Decorator implements ICopy{
    protected ICopy obj;
    public Decorator(ICopy obj){
        this.obj = obj;
    }
}
```

4. 定义目录复制装饰器 DirectCopy

```
class DirectCopy extends Decorator{
    public DirectCopy(ICopy copy){
        super(copy);
    }
    public void copy(String oldFolder,String newFolder)throws Exception{
        File file = new File(newFolder);        // 创建目标文件夹的 File 对象
        if (!file.exists())  file.mkdirs();     // 如果文件夹不存在，则创建文件夹
```

```
File oldFile = new File(oldFolder);  // 创建源文件夹的 File 对象
String[] files = oldFile.list();     // 获得源文件夹的文件列表
File tempFile = null;                // 创建存放文件的临时变量
for (int i = 0;i<files.length;i++){
    if (oldFolder.endsWith(File.separator)){ // 如果源文件夹以文件分隔符
                                             // 结尾
        tempFile = new File(oldFolder+files[i]); // 则直接连接文件名创建
                                                 // 临时文件对象
    }else{
        tempFile = new File(oldFolder+File.separator+files[i]);
    }
    if (tempFile.isFile()){                  // 临时文件对象是文件
      obj.copy(tempFile.getAbsolutePath(), newFolder+"/"+tempFile.
getName());//调已有拷贝构件
    }
    if (tempFile.isDirectory()){             // 临时文件对象是子文件夹
        // 递归调用拷贝方法
        copy(oldFolder+"/"+files[i],newFolder+"/"+files[i]);
    }
  }
 }
}
```

成员变量 obj 代表已有具体构件对象。目录拷贝采用了递归算法，对每个文件利用 obj 完成了拷贝功能。

5. 主界面类 MyFrame

```
class MyFrame extends JFrame{
    private String directSrc;
    private String directDest;
    JTextField fromtxt = new JTextField(60);
    JTextField totxt = new JTextField(60);
    public void init(){
        setLayout(null);
        JButton frombtn = new JButton("选择源目录");
        JButton tobtn = new JButton("选择目标目录");
        JButton btn = new JButton("开始拷贝");
        fromtxt.setEnabled(false); totxt.setEnabled(false);
        frombtn.setBounds(20, 30, 100, 30);
        tobtn.setBounds(20, 70, 100, 30);
        btn.setBounds(200, 110, 100, 30);
        fromtxt.setBounds(140,30,300,30);
        totxt.setBounds(140,70,300,30);
        add(frombtn);add(fromtxt);
        add(tobtn);add(totxt);
        add(btn);

        frombtn.addActionListener(new ActionListener(){ //选择源目录
            public void actionPerformed(ActionEvent e){
                directSrc = getDirect();fromtxt.setText(directSrc);
            }
        });
```

```
tobtn.addActionListener(new ActionListener(){    //选择目标目录
    public void actionPerformed(ActionEvent e){
        directDest = getDirect();totxt.setText(directDest);
    }
});
btn.addActionListener(new ActionListener(){ //目录拷贝按钮响应
    public void actionPerformed(ActionEvent e){
        ICopy obj = new Copy();                    //定义单文件拷贝具体构件
        ICopy obj2 = new DirectCopy(obj); //定义目录拷贝具体装饰器
        try {
            obj2.copy(directSrc, directDest);
        }
        catch(Exception ee) {ee.printStackTrace();}
    }
});
setSize(460,180); setResizable(false);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
}
private String getDirect(){
    JFileChooser fc = new JFileChooser();
    //设置对话框仅目录显示
    fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
    int ret = fc.showDialog(this, "请选择目录");
    if(ret == JFileChooser.APPROVE_OPTION)
        return fc.getSelectedFile().getAbsolutePath();
    return null;
}
public static void main(String[] args) {
    new MyFrame().init();
}
}
```

装饰模式调用的位置在 init()方法中。单击“开始拷贝”按钮时，事件响应的匿名类代码使用了装饰模式。另外，由于是目录拷贝，在 JFileChooser 对话框中必须仅显示各个目录，因此，必须对 JFileChooser 对象利用 setFileSelectionMode()，将对话框设置成目录显示状态。

6. 更强大的拷贝功能实现

到此为止，目录拷贝程序代码讲解完毕了。但根据具体构件 Copy 类的内容，无论将来怎样加以装饰，本质上实现的都是文件的拷贝。如果 IO 输入源是文件，输出源是 Socket，或者输入源是 URL，输出源是文件，那么 Copy 类就不好用了。那么能否屏蔽输入输出流的差异，实现输入输出流的拷贝呢？其实，从这句话中已经得出了答案，如下定义 ICopy 接口。

```
//重新定义 ICopy 接口
interface ICopy{
    void copy(InputStream in,OutputStream out)throws Exception;
}
```

也可以得出一般的结论：若操作是关于输入输出流的，那么，抽象构件定义的方法参数应是 InputStream 或 OutputStream 类型。具体构件流拷贝类也发生了变化，如下所示。

```
//重新定义流拷贝类 Copy
class Copy implements ICopy{
    public void copy(InputStream in, OutputStream out)throws Exception{
```

```
            while(in.available()>0){
                int value = in.read();
                out.write(value);
            }
        }
    }
```

这时忽然发现，装饰类 DirectCopy 出问题了：copy()方法定义与接口 ICopy 中定义的 copy()形式上不一致。从道理上来讲，不一致是正确的，因为目录可由字符串来标识其路径，而用 XXXStream 是无法表示目录特点的。因此，该方法定义不应该改变。但是这样的话，会出现编译错误，提示抽象方法 copy（InputStream,OutputStream）没有实现，怎么办呢？在抽象装饰类实现就可以了，如下所示。

```
//重新定义抽象装饰类 Decorator
abstract class Decorator implements ICopy{
    //其他代码同原 Decorator
    public void copy(InputStream in, OutputStream out)throws Exception{
        obj.copy(in, out);
    }
}
```

其实，copy()相当于为 Decorator 的子类提供了一个共享流拷贝方法，子类中可定义所需的各种 copy()方法，而无须考虑参数的形式。

```
//重新定义具体装饰类 DirectCopy
//仅把原代码中“if(tempFile.isFile()){…}”修改成如下所示即可
if (tempFile.isFile()) { // 临时文件对象是文件
    FileInputStream in = new FileInputStream(tempFile);
    FileOutputStream out = new FileOutputStream(newFolder + "/"
            + tempFile.getName());
    obj.copy(in, out);// 调用已有拷贝构件
    in.close();
    out.close();
}
```

主界面 MyFrame 类代码无需改变，执行结果和修改前的代码一致。

【例 11-2】 向数据库中导入大量数据及进度显示程序（主界面如图 11-5 所示），即当大量数据导入数据库的同时，进行进度显示。

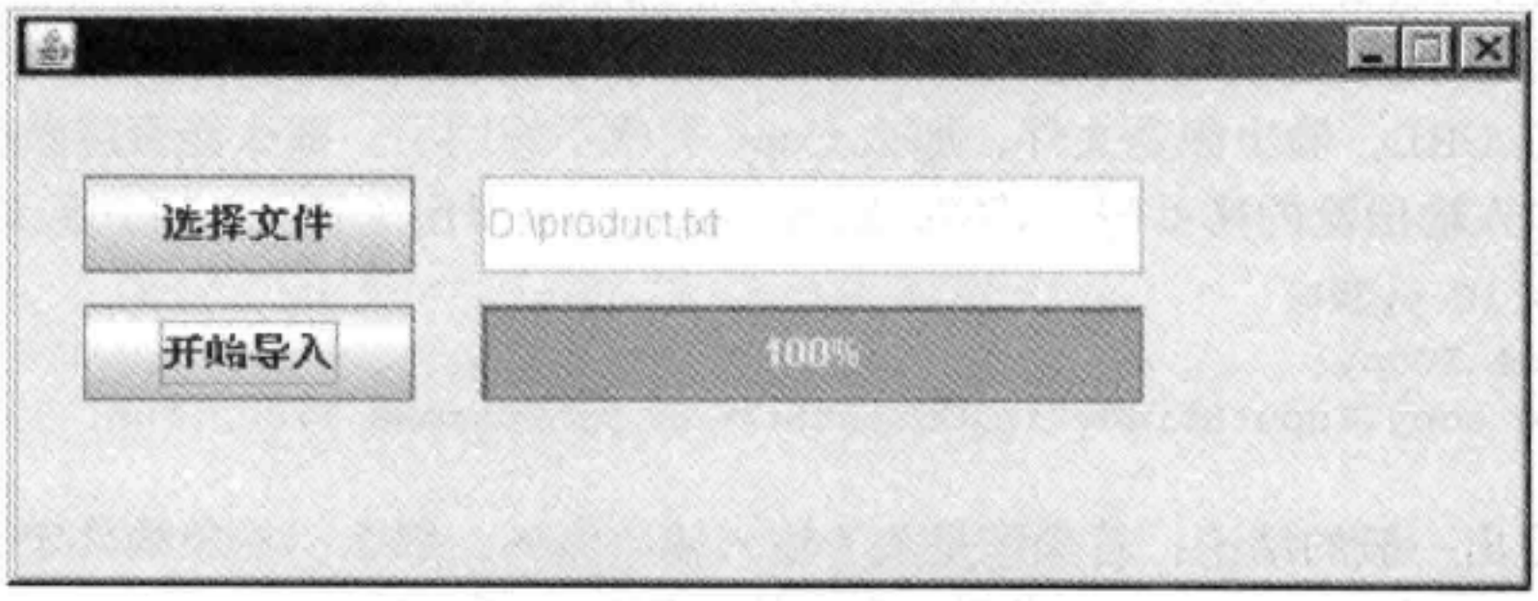


图 11-5　数据库数据导入及进度显示界面

执行该程序的前提条件是物理表（可任意）已经在数据库中建立，数据为空。大数据量文本格式如表 11-3 所示。

表 11-3　　大数据文本格式说明

product 20000 1000　铅笔　1.00 ……	• 第 1 行表示要导入的数据库表名 • 第 2 行表示数据记录数 • 第三行起一行一条记录，数据项间用\t 相隔

可以看出，数据库数据导入过程相当于具体构件功能，进度条显示相当于装饰功能。具体代码如下。

1. 定义数据导入抽象构件 IEntry

```
interface IEntry{
    int getCursor();        //当前游标位置
    int getTotal();         //总记录数目
    void entry(String strFile)throws Exception;
}
```

entry()是数据库数据导入方法，strFile 表示文本文件名称。getCursor()返回当前正在导入的文本记录的位置，getTotal()返回文本记录的总数目。

2. 定义数据库数据导入具体构件

```
class DbEntry implements IEntry{
    private String tabName;          //表名称
    private int total;               //记录总数
    private int cursor;              //当前记录位置
    public int getCursor(){
        return cursor;
    }
    public int getTotal(){
        return total;
    }
    public void entry(String strFile)throws Exception{
        FileReader in = new FileReader(strFile);
        BufferedReader in2 = new BufferedReader(in);
        tabName = in2.readLine();                          //读第 1 行数据获得表名
        total = Integer.parseInt(in2.readLine());          //读第 2 行数据获得记录总数

        DbProc dbobj = new DbProc();
        Connection conn = dbobj.connect();
        Statement stm = conn.createStatement();
        String s, strSQL, d[];
        while((s=in2.readLine())!=null){//第 3 行开始至文件尾是数据记录
            cursor ++;   //当前记录位置
            d = s.split("\t");
            strSQL = "insert into " +tabName+ " values(";
            for(int i=0; i<d.length; i++){
                if(i<d.length-1) strSQL += "'" +d[i]+ "',";
                else strSQL += "'" +d[i]+ "')";
            }
            stm.executeUpdate(strSQL);
        }
```

```
            stm.close(); conn.close();
            in2.close(); in.close();
        }
    }
```

由于文本数据量非常大，故采用把所有数据都读入缓冲区是不可取的。从 entry()方法可知，本示例算法是每读一条记录，就立即将其存入数据库，直至文本文件结束。当然也可以采取每次处理 *n*（*n*<<total）条记录或者其他有效的办法。

其实，当做大数据处理功能设计阶段的时候，就一定想到要进行进度控制，但在本功能类中又很难实现。因此可定义一些功能成员变量，让其他类来访问，以便做出更加精致的进度显示界面。所以本类中定义了总记录数目成员变量 total、当前正处理的记录位置 cursor。

3. 数据库数据导入抽象装饰器 Decorator

```
abstract class Decorator implements IEntry{
    protected IEntry obj;
    public Decorator(IEntry obj){
        this.obj = obj;
    }
    public int getCursor() {
        return obj.getCursor();
    }
    public int getTotal() {
        return obj.getTotal();
    }
    public void entry(String strFile) throws Exception {
        obj.entry(strFile);
    }
}
```

4. 数据库数据导入具体装饰器 Progress

```
class Progress extends Decorator implements Runnable{
    private JProgressBar bar;
    private String strFile ;
    private javax.swing.Timer timer;
    public Progress(IEntry obj, String strFile){
        super(obj);
        this.strFile = strFile;
        timer = new Timer(100, new ActionListener(){
            public void actionPerformed(ActionEvent e){
                if(getTotal()==0) return;
                bar.setValue(getCursor()*100/getTotal());
                if(getCursor()==getTotal()){
                    timer.stop();
                }
            }
        });
    }
    public void setBar(JProgressBar bar){
        this.bar = bar;
bar.setStringPainted(true);
    }
    public void run(){
        timer.start();
```

```
            try{
                super.entry(strFile);
            }catch(Exception e){e.printStackTrace();}
        }
    }
```

该类主要为数据库数据导入功能添加进度条显示装饰功能，因此要定义 JProgressBar 类型成员变量 bar。可能有许多读者看完代码后觉得很容易，但实际上并不是很容易想到，因为多数设计模式书籍装饰模式示例讲解中具体装饰器都很少涉及图形界面。所以，学习知识要灵活，不要生搬硬套，从语义出发就可以了。例如，对数据库数据导入增加什么功能？答案是增加进度条。那么，当然要在装饰类中定义进度条成员变量了。

由于涉及大量数据操作，要花费很多时间，因此 Progress 一定是线程类，一定是在 run()方法中调用 entry()数据导入方法。若 Progress 是非线程类，那么它一定和主图形用户界面处于同一线程，则当执行 entry()方法时，主界面的所有按钮等将都失效，直到 entry()执行结束，主界面上的按钮等才能进行事件响应。很明显，进度条数值显示也是一个线程，是由 Timer 定时器决定的。而且 run()方法中，代码 time.start()一定要在语句 super.entry(strFile)前面，若放在后面就不对了。

需要注意一点，不要在该类中创建 JProgressBar 对象，也就是不要有 new ProgressBar()语句。因为你不知道要把它放在图形界面的哪个位置，所以成员变量 bar 是由 setBar()方法确定的。

5. 主界面 MyFrame

```
public class MyFrame extends JFrame{
JTextField seltxt = new JTextField(60);
JProgressBar bar = new JProgressBar();
String strFile;
public void init(){
    setLayout(null);
    JButton selbtn = new JButton("选择文件");
    JButton btn = new JButton("开始导入");
    seltxt.setEnabled(false); seltxt.setEnabled(false);
    selbtn.setBounds(20, 30, 100, 30);
    btn.setBounds(20, 70, 100, 30);
    seltxt.setBounds(140, 30, 200, 30);
    bar.setBounds(140, 70, 200, 30);
    add(selbtn);add(seltxt);
    add(btn);add(bar);

    selbtn.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            JFileChooser fc = new JFileChooser();
            int ret = fc.showDialog(MyFrame.this, "请选择文件");
            if(ret == JFileChooser.APPROVE_OPTION){
                strFile =  fc.getSelectedFile().getAbsolutePath();
                seltxt.setText(strFile);
            }
        }
    });
    btn.addActionListener(new ActionListener(){
```

```
        public void actionPerformed(ActionEvent e){
            try{
                DbEntry obj = new DbEntry();
                Progress pg = new Progress(obj, strFile);
                pg.setBar(bar);
                new Thread(pg).start();
            }
            catch(Exception ee){ee.printStackTrace();}
        }
    });
    setSize(460,180); setResizable(false);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
    public static void main(String[] args)throws Exception {
        new MyFrame().init();
    }
}
```

装饰模式代码体现在 init()中为局部按钮 btn 添加消息响应的匿名类里。@ctionPerformed()方法最后一行一定用“new Thread(pg).start()”语句启动线程，若直接写 pg.run()就错误了。

【例 11-3】 Web 程序分页显示功能设计与实现。

分页显示是 Web 应用的常用功能，本示例旨在利用装饰模式构建通用分页显示功能类，单页显示作为具体构件，具体装饰类增加了控制条功能，包括“首页”、“上一页”、“下一页”、“尾页”等控制功能，具体代码如下。

1. 定义抽象显示构件 IShow

```
publicinterface IShow {
    int getPage();
    int getTotal();
    String show(HttpServletRequest req)throws Exception;
}
```

getPage()返回当前显示的页数，getTotal()返回该查询语句的总页数，show()获得显示当前页的 HTML 字符串。

2. 具体单页显示构件 Show

```
public class Show implements IShow {
    HttpServletRequest req;
    int pagesize = 10;//为了简化，每页默认显示 10 条记录
    int page;          //当前要显示的页号
    int pagetotal;     //总页数
    String strSQL;     //查询 SQL 语句
    String title[];    //字段显示说明
    String width[];    //字段显示宽度
    public int getPage() {return page;}          //返回当前页
    public int getTotal(){return pagetotal;} //返回总页数
    private void getInfo(){                      //从 HttpServletRequest 获得接收参数
        strSQL = req.getParameter("strsql"); //查询 SQL 语句
        page = Integer.parseInt(req.getParameter("page"));    //待显示的页数
        title = req.getParameterValues("title");              //获得字段说明数组
```

```
        width = req.getParameterValues("width");                    //获得字段宽度数组
    }
    public String show(HttpServletRequest req)throws Exception {
        this.req = req;
        getInfo();

        DbProc dbobj = new DbProc();
        Connection conn = dbobj.connect();
        //求该查询语句总记录数
        int pos = strSQL.indexOf("from");
        String subs = strSQL.substring(pos, strSQL.length());
        String s = "select count(*) " + subs;
        Statement stm = conn.createStatement();
        ResultSet rst = stm.executeQuery(s);
        int total = 0;
        if(!rst.next())
          total = 0;
        else
          total = rst.getInt(1);
        rst.close();
        //获得待显示页数对应的查询 sql 语句
        int pagetotal=total/pagesize;
        if(total % pagesize !=0)pagetotal++;
        if(page>pagetotal) page=pagetotal;
        int start = (page-1)*pagesize;
        s = strSQL + " limit " +start+ "," +pagesize;
        //查询并形成 html 字符串
        rst = stm.executeQuery(s);
        s = "<table border='1'>";
        //形成表头字符串 subhead
        String subhead = "<tr>";
        for(int i=0; i<title.length; i++){
            subhead += "<th width='" +width[i]+ "'>" +title[i]+ "</th>";
        }
        subhead += "</tr>";
        //形成表体内容字符串
        subs = "";
        while(rst.next()){
            subs += "<tr>";
            for(int i=1; i<=title.length; i++){
            subs += "<td>" +rst.getString(i)+ "</td>";
            }
            subs += "</tr>";
        }
        s += subhead;
        s += subs;
        s += "</table>";
        rst.close();stm.close();
            return s;
    }
}
```

一般来说，只要知道待查询的 sql 语句、待显示的页数、每列显示信息的文字说明及单元格宽度这四种具体信息，就能画出相应的表格。对应这四种信息定义了四个成员变量：strSQL、page、title[]、width[]，它们的具体值是通过 URL 传过来，利用 HttpServletRequest 解析的，具体见 getInfo()方法。

show()方法主要描述了形成通用表格字符串的过程。利用“select count(*).....”求出该查询的总记录数 total，根据每页记录数 pagesize 可算出当前 page 页对应的记录开始位置 start，获得当前页对应的具体查询语句“select ..... limit start,pagesize”，进而获得查询记录集 rst。基础数据获得完毕后，利用 title[]、width[]形成表头字符串 subhead，遍历 rst 获得表体内容字符串 subs，两者合并最终形成完整的表格 HTML 结果字符串。

3. 抽象分页显示装饰器 Decorator

```
public abstract class Decorator implements IShow {
    protected IShow obj;
    public Decorator(IShow obj){
        this.obj = obj;
    }
    public String show(HttpServletRequest req) throws Exception {return obj.
        show(req);}
    public int getPage() {return obj.getPage();}
    public int getTotal() {return obj.getTotal();}
}
```

4. 具体分页装饰器 PageShow

```
public class PageShow extends Decorator {
    public PageShow(IShow obj){super(obj);}
    public String ctrlShow(){
        String s = "<div>"+
            "<a title='first' href='javascript:void(0)' onclick='go(this)'>首页
            </a>"+
            "    "+
            "<a title='prev' href='javascript:void(0)' onclick='go(this)'>上一
            页</a>"+
            "    "+
            "<a title='next' href='javascript:void(0)' onclick='go(this)'>下一
            页</a>"+
            "    "+
            "<a title='tail' href='javascript:void(0)' onclick='go(this)'>尾页
            </a>"+
            "    "+
            "转到第";
        String subs = "<select id='selctl' onchange='gosel(this)'>";
        for(int i=1; i<=getTotal(); i++){
            if(i==getPage())
                subs += "<option selected value='"+i+"'>"+i+"</option>";
            else
                subs += "<option value='"+i+"'>"+i+"</option>";
        }
        subs += "</select></div>";
        return s + subs;
    }
```

```
    public String show(HttpServletRequest req)throws Exception{
        String s = super.show(req);
        s += ctrlShow();
        return s;
    }
}
```

可以看出，分页显示增加了“首页”、“上一页”、“下一页”、“尾页”超链接及一个页号选择标签。4 个超链接标签 title 属性值不同，对应的 javascript 方法相同，都是 go()，在 go() 中利用 title 属性值判断究竟执行的是哪一个功能。选择标签 select 的 id 值是 selctl，javascript 响应方法是 gosel()。

到此为止，分页显示的装饰模式功能类代码已经完备了。由于 Web 程序涉及知识较多，为了使读者有一个完整的认识，以显示例 11-2 中的产品表 product 为例，还需要以下步骤。

5. 定义产品表分页显示 servlet 类 Product，URL 为 product

```
public class Product extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public Product() { }
    protected void doPost(HttpServletRequest req, HttpServletResponse rep)
            throws ServletException, IOException {
        rep.setContentType("text/html;charset=utf-8");
        req.setCharacterEncoding("utf-8");
        try{
            IShow obj = new Show();
            IShow obj2 = new PageShow(obj);
            String s = obj2.show(req);
            rep.getWriter().print(s);
        }catch(Exception e){e.printStackTrace();}
    }

}
```

该类比较简单，在 try 块中包含了装饰模式的调用代码。

6. 一个简单的测试文件 test.jsp

```
<html>
<script type="text/javascript">
    var page = 1;                          //当前显示页
    var xmlHtpRq = new ActiveXObject("Microsoft.XMLHTTP");        //ajax 变量
    function go(e){          //“首页-尾页”控制条响应方法
        var max = document.getElementById("selctl").options.length;
        if(e.title=="first") page=1;
        if(e.title=="prev") page--;
        if(e.title=="next") page++;
        if(e.title=="tail") page = document.getElementById("selctl").options.length;
        if(page<1) page=1;
        if(page>max) page=max;
        productShow();
    }
    function go_sel(e){                    //选择标签，选择显示页响应方法
        page = parseInt(e.value); go();
    }
    function productShow(){
```

```
var strSQL = "select * from product";          //查询具体 SQL 语句
var t = new Array("NO","Name","Price");        //查询列名称数组
var w = new Array(100,200,50);                 //列显示宽度数组
var i=0,title="",width="";
for(i=0; i<t.length-1; i++)
    title += "title="+t[i]+"&";
title += "title="+t[i];
for(i=0; i<w.length-1; i++)
    width += "width="+w[i]+"&";
width += "width="+w[i];
var url = "product?page="+page+"&strsql="+strSQL+"&"+title+"&"+width;
        xmlHtpRq.open("post", url, true);
        xmlHtpRq.onreadystatechange = productShow_state;
        xmlHtpRq.send(null);
    }
    function productShow_state(){
        if(xmlHtpRq.readyState == 4){
            if(xmlHtpRq.status == 200){
                var obj = document.getElementById("show");
                obj.innerHTML = xmlHtpRq.responseText;
            }
        }
    }
</script></head>
<body>
<div><input type="button" value='显示产品表' onclick="productShow()"/></div>
<div id="show"></div>
</body>
</html>
```

执行页面如图 11-6 所示。

当单击“显示产品表”按钮时，执行 productShow()方法，在该方法中形成的 URL 包括查询 sql 语句、当前显示页数、字段显示说明及宽度数组。通过 Ajax 技术调用 servlet 类 Product，该类把结果返回 Ajax 接收方法 productShow_state()，在该方法内把接收结果动态填充在 id 为 show 的 标签内。

当运行“首页”、“上一页”、“下一页”、“尾页”超链接时先执行 go()，再执行 productShow()；当选中选择标签某一页时，先运行 gosel()，再运行 productShow()。

图 11-6 分页显示页面结果

限于篇幅，本示例在某些方面通用性还不强，就不多加讨论了，但可提出来与读者共忖：①go()及 gosel()方法中最终去向都是 productShow()，这决定了仅能显示 product 表，如何改造 javascript 方法，使之能适应任意 sql 语句的查询？②能否进一步对 PageShow 类进行装饰，使之不仅有分页显示功能还有删改功能？

第12章 组合模式

12.1 问题的提出

我们研究的问题有许多树型结构的问题，例如文件结构，如图 12-1 所示。

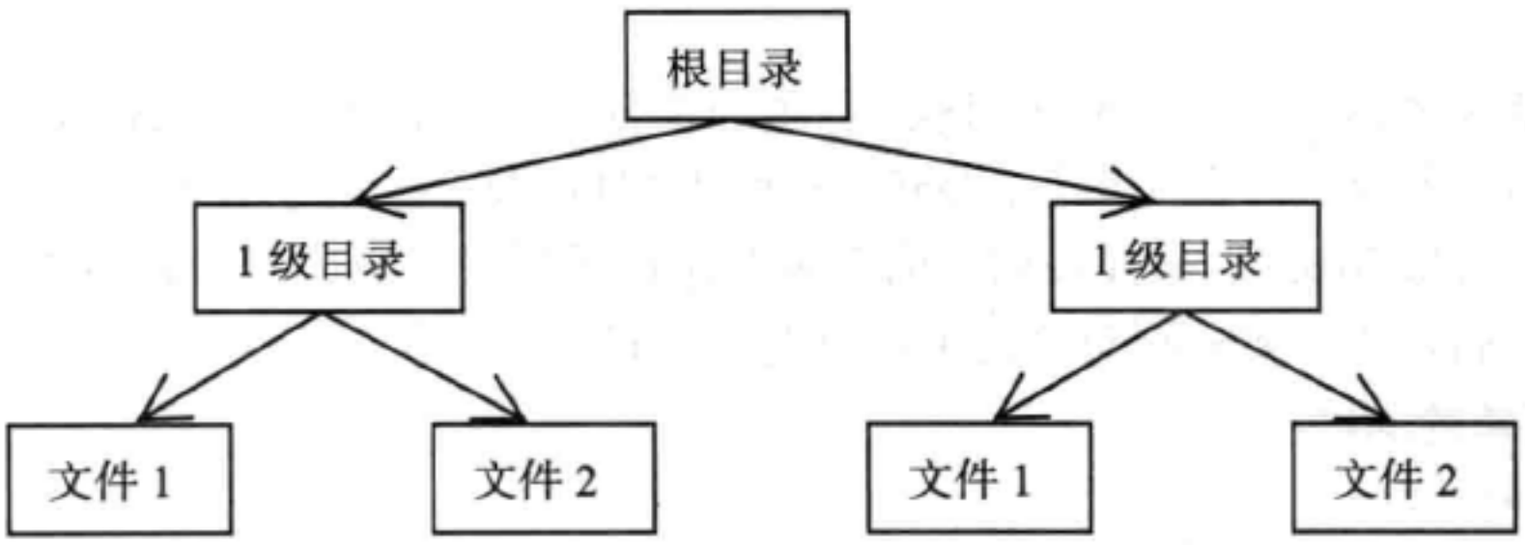


图 12-1 文件树型结构示例图

例如，要用程序创建文件树型结构，为了验证正确与否，还要在控制台上输出从某目录开始的所有文件信息。

很明显，文件树型结构节点可分为两类：一类是文件叶子节点，无后继节点；另一类是中间目录节点，有后继节点。因此可得具体代码如下。

1. 文件节点类 FileLeaf

```
class FileLeaf{
    String fileName;
    public FileLeaf(String fileName){
        this.fileName = fileName;
    }
    public void display(){
        System.out.println(fileName);
    }
}
```

2. 中间目录节点类 DirectNode

```
class DirectNode{
    String nodeName;
    public DirectNode(String nodeName){
        this.nodeName = nodeName;
    }
```

```
ArrayList<DirectNode> nodeList = new ArrayList();    //后继子目录集合
ArrayList<FileLeaf> leafList = new ArrayList();      //当前目录文件集合
public void addNode(DirectNode node){          //添加下一级子目录
    nodeList.add(node);
}
public void addLeaf(FileLeaf leaf){            //添加本级文件
    leafList.add(leaf);
}
public void display(){                         //从本级目录开始显示
    for(int i=0; i<leafList.size(); i++){      //先显示文件
        leafList.get(i).display();
    }
    for(int i=0; i<nodeList.size(); i++){   //再显示子目录
        System.out.println(nodeList.get(i).nodeName);
        nodeList.get(i).display();
    }
}
}
```

该节点类表明本级目录一定有子目录，可能有文件。因此定义了两个 ArrayList 类型的成员变量，nodeList 表示子目录的集合，leafList 表明文件的集合。由于有了这两个集合，因此有两个添加方法 addNode()、addLeaf()，前者表明添加子目录对象，后者表明添加文件对象。同理，在 display()方法中要分别对这两个集合遍历并输出结果。

3. 一个简单的测试类

```
public class Test {
    public static void createTree(DirectNode node){
        File f = new File(node.nodeName);
        File f2[] = f.listFiles();
        for(int i=0; i<f2.length; i++){
            if(f2[i].isFile()){
                FileLeaf l = new FileLeaf(f2[i].getAbsolutePath());
                node.addLeaf(l);
            }
            if(f2[i].isDirectory()){
                DirectNode node2 = new DirectNode(f2[i].getAbsolutePath());
                node.addNode(node2);
                createTree(node2);   //递归调用生成树结构
            }
        }
        public static void main(String[] args) {
            DirectNode start = new DirectNode("d://data"); //创建该目录的树型结构集合
            createTree(start);   //创建过程
            start.display();     //显示过程，验证创建是否正确
        }
}
```

由于是树型结构，因此 createTree()方法一定是递归调用的。那么有没有更好的方式完成上述功能，结构更优雅呢？组合模式就是解决树型结构问题强有力的设计工具。

12.2 组合模式

由图 12-1 可以知道，根目录是由两个子目录组成的，第一级子目录由两个文件组成，第二级子目录由两个文件组成，因此树型形式也可以称作组合形式。在 12.1 节中，把节点分为叶子节点和目录节点，它们是孤立的。其实只要思维再前进一步，就会发生质的变化。那就是把叶子节点与目录节点都看成相同性质的节点，只不过目录节点后继节点不为空，而叶子节点后继节点为 null。这样就能够对树型结构的所有节点执行相同的操作，这也是组合模式的最大特点。采用组合模式修改 12.1 节中的功能，具体代码如下。

1. 定义抽象节点类 Node

```
abstract class Node{
    protected String name;
    public Node(String name){
        this.name = name;
    }
    public void addNode(Node node)throws Exception{
        throw new Exception("Invalid exception");
    }
    abstract void display();
}
```

该类是叶子节点与目录节点的父类，节点名称是 name。其主要包括两类方法：一类方法是所有节点具有相同形式、不同内容的方法，这类方法要定义成抽象方法，如 display()；另一类方法是目录节点必须重写，叶子节点无需重写的方法，相当于为叶子节点提供了默认实现，如 addNode()方法，因为叶子对象没有该功能，所以可以通过抛出异常防止叶子节点无效调用该方法。

2. 文件叶子节点类 FileNode

```
class FileNode extends Node{
    public FileNode(String name){
        super(name);
    }
    public void display(){
        System.out.println(name);
    }
}
```

该类是 Node 的派生类，仅重写 display()方法即可。

3. 目录节点类 DirectNode

```
class DirectNode extends Node{
    ArrayList<Node> nodeList = new ArrayList();
    public DirectNode(String name){
        super(name);
    }
    public void addNode(Node node)throws Exception{
        nodeList.add(node);
    }
    public void display(){
```

```
            System.out.println(name);
            for(int i=0; i<nodeList.size(); i++){
                nodeList.get(i).display();
            }
        }
    }
```

该类从 Node 抽象类派生后，与原 DirectNode 类相比，主要有以下不同：①由定义两个集合类成员变量转为定义一个集合类成员变量 nodeList；②由定义两个添加方法转为定义一个添加方法 addNode()；③display()方法中，由两个不同元素的循环转为一个对相同性质节点 Node 的循环。也就是说，原 DirectNode 中不论是定义成员变量、成员方法，还是方法内部的功能，都要实时考虑叶子节点、目录节点的不同性，因此它的各种定义一定是双份的。而组合模式中认为叶子节点、目录节点是同一性质的节点，因此与原 DirectNode 类对比，它的各种定义工作一定是减半的，也易于扩充。

4. 一个简单的测试类

```
public class Test {
    public static void createTree(Node node)throws Exception{
        File f = new File(node.name);
        File f2[] = f.listFiles();
        for(int i=0; i<f2.length; i++){
            if(f2[i].isFile()){
                Node node2 = new FileNode(f2[i].getAbsolutePath());
                node.addNode(node2);
            }
            if(f2[i].isDirectory()){
                Node node2 = new DirectNode(f2[i].getAbsolutePath());
                node.addNode(node2);
                createTree(node2);
            }
        }
    }
    public static void main(String[] args)throws Exception {
        Node start = new DirectNode("d://data");
        createTree(start);
        start.display();
    }
}
```

该类与原测试类相近。可以看出不论是对文件还是目录节点，都是通过添加 Node 对象节点形成树型结构的。

通过该示例，可得组合模式更一般的 UML 类图，如图 12-2 所示。共包括以下三种角色。

- 抽象节点：Node，是一个抽象类（或接口），定义了个体对象和组合对象需要实现的关于操作其子节点的方法，如 add()、remove()、display()等。
- 叶节点：Leaf，从抽象节点 Node 派生，由于本身无后继节点，其 add()等方法利用 Node 抽象类中相应的默认实现即可，只需实现与自身相关的 remove()、display()等方法即可。
- 组合节点：Component，从抽象节点 Node 派生，包含其他 Composite 节点或 Leaf 节点的引用。

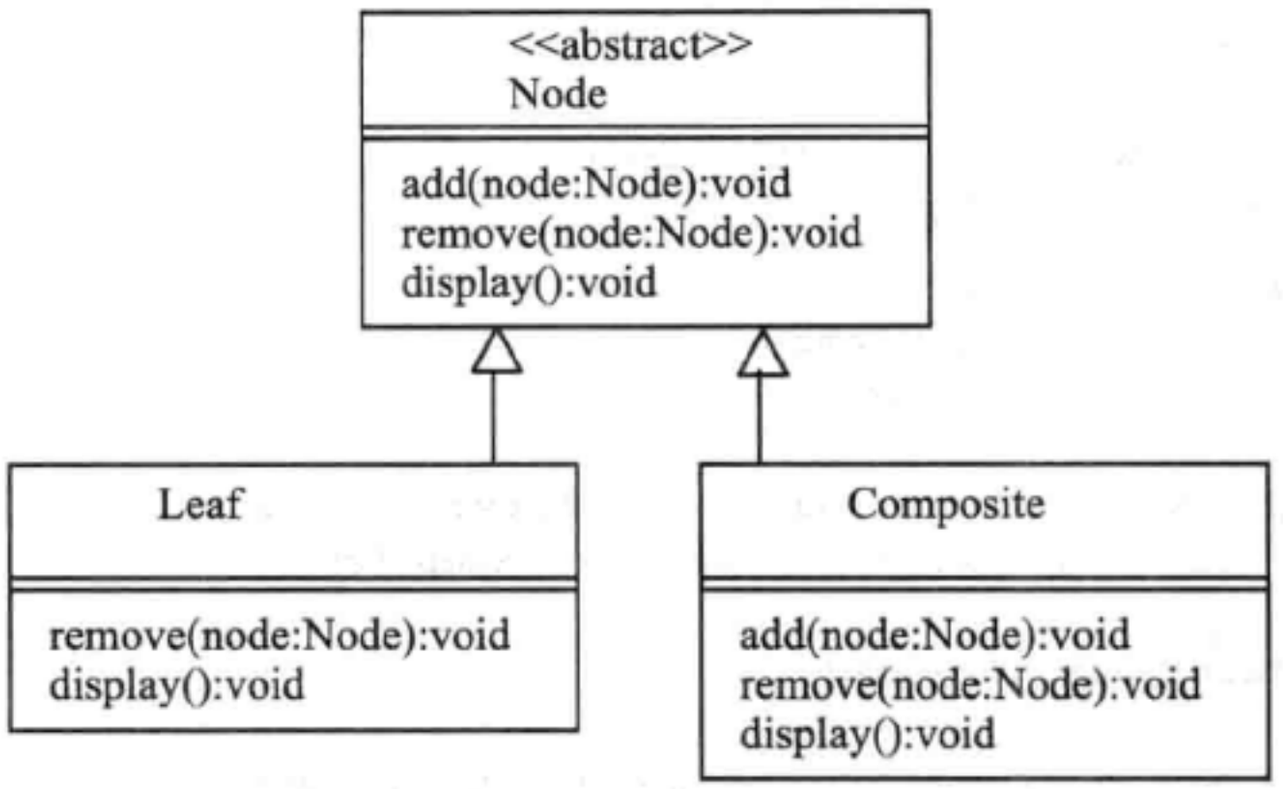


图 12-2 组合模式 UML 类图

总之，若某应用可形成树型结构，而且形成树型结构后可对叶节点及中间节点进行统一的操作，那么采用组合模式构建应用功能是一个比较好的选择。

12.3 深入理解组合模式

12.3.1 其他常用操作

当形成树型结构后，那么数据结构中一些常用操作都可以实现编程了。例如在 12.2 节中示例的基础上要求实现以下功能：返回父节点，返回子女节点，返回兄弟节点。其具体代码如下。

1. 定义抽象节点 Node

```
abstract class Node{
    protected Node parent=null;          //定义父节点
    public void setParent(Node parent){
        this.parent = parent;
    }
    public Node getParent(){
        return parent;
    }
    public Node[] getBrothers(){         //获得兄弟节点
DirectNode parent = (DirectNode)getParent();
if(parent == null)
    return null;
int size = parent.nodeList.size();
if(size==1)
    return null;
Node nodes[] = new Node[size-1];
for(int i=0; i<size; i++){
    if(parent.nodeList.get(i)==this)
        continue;
    nodes[i] = parent.nodeList.get(i);
```

```
        }
        return nodes;
    }
    public abstract Node[] getChilds();
    //下面代码同 12.2 节中代码
    protected String name;
        public Node(String name) {
            this.name = name;
        }
        public void addNode(Node node) throws Exception {
            throw new Exception("Invalid exception");
        }
        abstract void display();
}
```

该类定义了父节点 Node 变量 parent，如何为其赋值并不在本类中，是在子类实现的。获得子类对象 getChilds()是抽象方法，表明要在子类具体实现。获得兄弟节点 getBrothers()是普通方法为子类所共享，子类无需实现。其主要思路是：获得该节点的父类节点，对于叶子及目录节点，其父节点均是目录节点，因此有强制转换语句“DirectNode parent=(DirectNode) getParent”。根据 parent 对象，可知它有哪些子对象，再去除当前节点对象，就可得兄弟对象数组了。

2. 文件叶子节点类 FileNode

```
class FileNode extends Node{
// 下面代码同 12.2 节中代码
    public FileNode(String name) {
        super(name);
    }
    public void display() {
        System.out.println(name);
    }
    //新增代码
    public Node[] getChilds(){
        return null;
    }
}
```

文件节点没有子节点，因此返回 null 值。

3. 目录节点类 DirectNode

```
class DirectNode extends Node{
    ArrayList<Node> nodeList = new ArrayList();
    public DirectNode(String name) {
        super(name);
    }
public void display() {
    System.out.println(name);
    for (int i = 0; i < nodeList.size(); i++) {
        nodeList.get(i).display();
    }
}
  // 上面代码同 12.2 节中代码
public void addNode(Node node)throws Exception{
```

```
        nodeList.add(node);
        node.setParent(this);        //node 的父节点是 this 节点
    }
    public Node[] getChilds(){
        if(nodeList.size()==0)
            return null;
        Node nodes[] = new Node[nodeList.size()];
        for(int i=0; i<nodeList.size(); i++){
            nodes[i] = nodeList.get(i);
        }
        return nodes;
    }
}
```

在 addNode()方法中添加节点的同时，设置父节点对象。获得子对象 getChilds()算法简单，在此就不再论述了。

12.3.2 节点排序

有效树型数据结构比较方便查询，那么什么是有效数据结构呢？按每一层节点关键字排好序的树就是一种有效的树型数据结构。例如英文字典树，若第一层按“a，b，……，z”排好序，那么查“about”单词只需按第一个分支“a”开始向下查，其他分支根本无需查。仍以 12.2 节中目录树为例，按节点字符串名称升序排列，其代码如下。

```
abstract class Node{ /*同 12.2 节*/     }
class FileNode extends Node{      /*同 12.2 节*/}
class DirectNode extends Node{
Set<Node> nodeList = new TreeSet(new Comparator(){
    public int compare(Object obj, Object obj2) {
        Node one = (Node)obj;
        Node two = (Node)obj2;
        return one.name.compareTo(two.name);
    }
});
public DirectNode(String name){
    super(name);
}
public void addNode(Node node)throws Exception{
    nodeList.add(node);
}
public void display(){
    System.out.println(name);
    Iterator<Node> it = nodeList.iterator();
        while(it.hasNext()){
            Node node = it.next();
            node.display();
        }
    }
}
public class Test {  /*测试类同 12.2 节*/ }
```

成员变量类型利用 Set 替代了 ArrayList，而且用的是 TreeSet。TreeSet 要求必须定义插入 Node 对象规则，该规则是由实现 Comparator 接口的具体类决定的。本例中是用匿名类 new

Comparator()实现的，可知遍历 TreeSet 时是按节点名称升序输出节点内容的。

12.4 应 用 探 究

【例 12-1】 英汉字典查询。

示例功能是根据英文查询出对应的汉语解释。毫无疑问，利用字典树能提高查询速度。以图 12-3 加以说明。

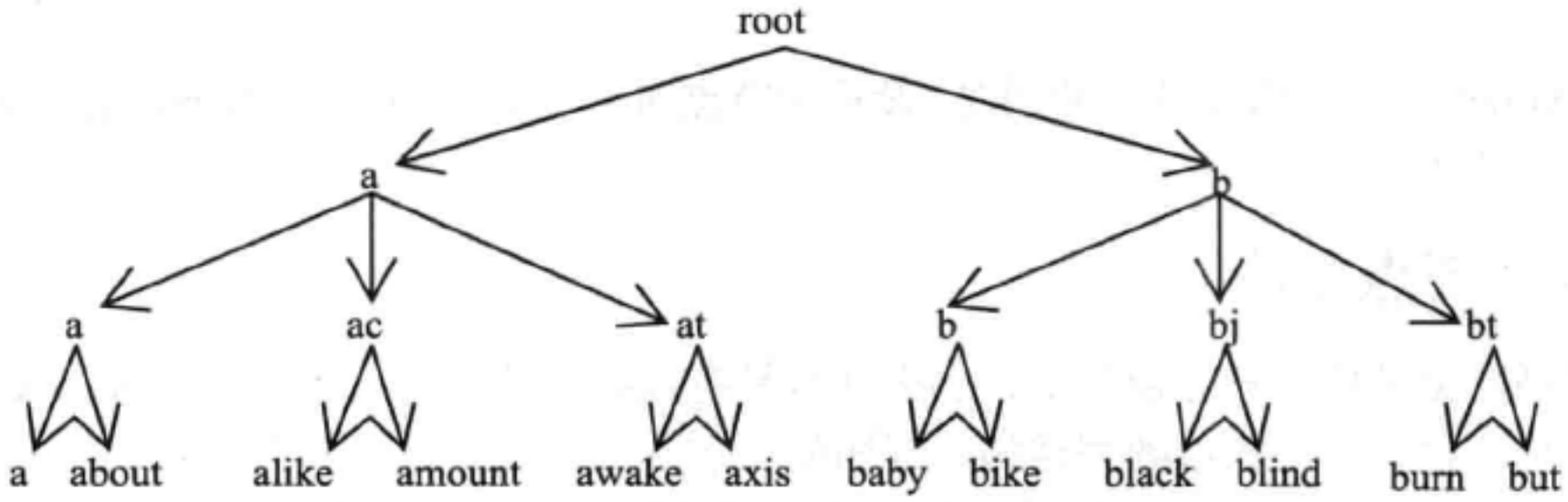


图 12-3 字典树说明

该字典树的特点：每层的节点值都是递增的，子节点的值都大于等于父节点的值，但是小于父节点右侧的最近的兄弟节点的值。例如，用表意形式讲，[a, about]≥[a]，[a,about]<[b]；另一个特点是所有英文单词都是叶子节点，中间节点都是分支节点。采用组合模式的具体代码如下。

1. 单词类 Word

```
class Word{
    String english;
    String chinese;
    public Word(String english, String chinese){
        this.english=english; this.chinese=chinese;
    }
}
```

这是关于单词的基础类，现在比较简单，仅有英文单词及一个汉语解释。当然，可以增加功能，如多种解释、短语、例句等，只要保证对应一个英文单词即可。

2. 组合模式抽象节点类 Node

```
abstract class Node{
    String key;        //节点关键值
    Word w = null;
    Node parent = null;//父节点
    public Node(String key, Word w){
        this.key = key;
        this.w = w;
    }
    public Node getParent() {
        return parent;
    }
    public void setParent(Node parent) {
```

```
        this.parent = parent;
    }
    public void addNode(Node node)throws Exception{
        throw new Exception("Invalid exception");
    }
}
```

组合模式把叶子节点和中间节点都看作同一性质的节点。一般来说，所有节点都有一个关键字符串作为标识，因此该类定义了字符串成员变量 key；又由于叶子节点是 Word 对象，所以又定义了 Word 类型成员变量 w。当然，对叶子及中间节点 w 值的处理方法一定是不同的，不同在哪里？希望读者带着这样的问题往下看。

3. 单词叶子节点类 WordNode

```
class WordNode extends Node{
    public WordNode(String english, Word w){
        super(english, w);
    }
}
```

对于单词节点类而言，Word 对象必须设置。

4. 中间比较节点类 MidNode

```
class MidNode extends Node{
    Set<Node> nodeList = new TreeSet(new Comparator(){
        public int compare(Object obj, Object obj2) {
            Node one = (Node)obj;
            Node two = (Node)obj2;
            return one.key.compareTo(two.key);
        }
});
public MidNode(String key){
    super(key,null); //Word 对象设置为 null
}
public void addNode(Node node)throws Exception{
    nodeList.add(node);
    node.setParent(this);
}
public Node get(int pos){  //返回该节点的第 pos 个子节点，从 0 开始
    Node node = null;
    Iterator<Node> it = nodeList.iterator();
    for(int i=0; i<=pos; i++){
        node = it.next();
    }
    return node;
  }
}
```

由于中间节点只需要一个键值，无需 Word 对象，因此在构造方法中利用“super(key,null)”将 Word 对象置为 null 即可。

到此为止，组合模式的功能类编制完毕了。应用这些类如何编制英汉翻译呢？笔者认为，还要编制一个字典管理类 Dictionary，里面至少要包含创建字典树及查询两个方法，其代码如下。

```
class Dictionary{
```

```
    Node root = new MidNode("root");
    public void create()throws Exception{ /*代码描述见下文*/ }
    void search(String english){ /*代码描述见下文*/ }
    public static void main(String[] args)throws Exception {
        Dictionary dt = new Dictionary();
        dt.create();
        dt.search("axis");dt.search("axis"); dt.search("blind");
    }
}
```

对于字典类来说，根节点对象是重要的，因此把它定义为成员变量 root，有了它就可以遍历字典树了。create()、search()方法代码较长，下面分别加以叙述。

```
public void create()throws Exception{
    String one[] = {"a","b"};
    String two[][] = {{"a","ac","at"},{"b","bj","bt"}};
    String three[][][]={{{"a","about"},{"alike","amount"},{"awake","axis"}},
            {{"baby","bike"},{"black","blind"},{"burn","but"}}};
    String china[][][]={{{"一个","关于"},{"象","数量"},{"醒","轴"}},
            {{"婴儿","自行车"},{"黑","瞎"},{"燃烧","但是"}}};
    Node parent = null, parent2=null, child=null;
    for(int i=0; i<one.length; i++){
        child = new MidNode(one[i]);
        root.addNode(child);                    //添加第 1 层子节点
    }
    for(int i=0; i<one.length; i++){
        parent = ((MidNode)root).get(i);  //获得第 1 层结点
        for(int j=0; j<two[i].length; j++){
            child = new MidNode(two[i][j]);
            parent.addNode(child);      //第 1 层节点添加第 2 层节点
        }
    }
    for(int i=0; i<one.length; i++){
        parent = ((MidNode)root).get(i);  //获得第 1 层节点
        for(int j=0; j<two[i].length; j++){
            parent2 = ((MidNode)parent).get(j); //获得第 2 层节点
            for(int k=0; k<three[i][j].length; k++){
                Word w = new Word(three[i][j][k],china[i][j][k]);
                WordNode wn = new WordNode(three[i][j][k],w);
                parent2.addNode(wn);  //添加第 3 层节点
            }
        }
    }
```

为了方便，此方法内创建的树型结构与图 12-3 一致，one、two、three 数组分别对应第 1、2、3 层节点，three 数组也是英语单词节点。china 数组与 three 数组一一对应，是英文单词的汉语解释。本方法就是将 one、two、three、china 数组转化成树型节点信息，是由 3 个独立的循环完成的。当然，更一般的形式是从文件导入，或从界面以某种输入方式等，方法多种多样。总之，利用程序创建完备的树型结构非常有挑战性，读者要多加思考，从中也更能体会出程序的乐趣。

根据英文查询汉语解释的方法 search()如下所示。

```
void search(String english){
    Node parent = root;          //从根节点向下开始查询
    Set<Node> s;
    Node cur=null, next=null;
    Iterator<Node> it;
    while(true){
        s = ((MidNode)parent).nodeList;     //获得节点的子节点集合
        it = s.iterator();
        cur = it.next();                    //设置当前节点
        while(it.hasNext()){
            next = it.next();               //获得下一个节点
            if(english.compareTo(next.key)<0)   //若英语单词小于 next 节点关键字，则
                                                //退出
                break;
            cur = next;
        }
        s = ((MidNode)cur).nodeList;        //待查英文单词一定在 cur 节点子列表中
        it = s.iterator();
        if(it.next() instanceof WordNode)   //若已查到单词节点，则退出
            break;
        parent = cur;
    }
    s = ((MidNode)cur).nodeList;            //所查单词一定在 cur 节点的下一级节点上
    it = s.iterator();                      //该下一级节点是单词节点
    boolean bmark = false;
    while(it.hasNext()){
        Node tmp = it.next();
        if(tmp.key.equals(english)){
            System.out.println(english+"--->"+tmp.w.chinese);
            bmark = true;
            break;
        }
    }
    if(bmark==false)
        System.out.println(english+"没有恰当的翻译");
}
```

先用示例来说明查询算法，例如在图 12-3 中查询单词 amount。遍历 root 子节点，第 1 个节点 a<amount，第 2 个节点 b>amount，所以 amount 单词一定在 a 节点下。遍历 a 子节点，第 1 个节点 a<amount，第 2 个节点 ac<amount，第 3 个节点 at>amount，所以 amount 单词一定在 ac 节点下，易判定 ac 子结点已是单词节点，故所查单词就在 ac 的下一级节点上，遍历下一级节点，即可查到翻译结果。综合上述，得查询算法文字描述如表 12-1 所示。

表 12-1　　英文单词查询算法

```
search(english)
1    parent ←字典根节点 root
2    while(true)
3        s ← parent 节点的子节点集和
4        cur ←s 中第一个元素
5        遍历 s 中其余节点
```

续表

6	next←当前遍历值
7	若 english<next 节点关键字 则转 9
8	cur←next, 转 5
9	s ← cur 节点的子结点集和
10	若 s 已是单词节点， 则转 13
11	parent ← cur，转 2
12	end while
13	遍历 cur 子节点集合，若某节点关键值等于 english, 则输出对应的汉语解释。

5. 一个测试类

```
public class Test {
    public static void main(String[] args)throws Exception {
        Dictionary dt = new Dictionary();
        dt.create();
        dt.search("myaxis");
        dt.search("axis");
    }
}
```

可能读者有疑问:实现本题功能根本无需树型结构,只要形成 ArrayList<Node>升序集合,再利用 BinarySearch()方法就能实现单词翻译功能。诚然,单纯从功能角度来说是没有问题的。从拓展角度来说有以下不足:①若在已排好序的 ArrayList 中新增一个单词,将会涉及数组元素的移动,而树型结构则会方便添加元素,只不过目前树型结构的基础代码还有待完善;②ArrayList 无法形成一个有效的电子书,而树型结构就相当于电子书的目录,易于形成电子书。当然,还有如某区间元素特征统计等,树型结构都显示出了它的优越性。

【例 12-2】 XML 文件解析程序。

XML 可扩展标注语言正被迅速地运用于各领域,已成为与平台、语言和协议无关的格式描述和交换数据的应用标准。操纵 XML 的 API 有很多种,如 DOM、SAX、JDOM 等。本示例抛开这些专业的解析工具,自定义算法解析 XML 文件。先看一个 XML 文件示例,如表 12-2 所示。

表 12-2 XML 文件示例

```
<root>
    <a name=a prop1=a2>
        <b name=b1 prop1=b11>bvalue</b>
        <b name=b2>bvalue2</b>
        <c>cvalue</c>
    </a>
    <a2>avalue2</a2>
</root>
```

很明显,XML 各标签间关系符合树型节点的特点,因此组合模式是解析 XML 文件的有力工具。为了解析方便,做以下约束。

- 叶节点和中间节点的形式固定,这是最重要的一点。对于叶节点而言,标签在一行中一定是封闭的。即若有<>,则一定有</>,如示例中的<b>和<a2>标签。对于中间节点而言,标签一定是不封闭的,一定是多行后才封闭的,如示例中的<root>和<a>标签。

- 对于叶节点，有 3 个重要域：名字、属性集、值。如示例中第 1 个<b>标签，标签名是 b，值是 bvalue，有两个属性，属性 name 对应值是 b1，属性 prop1 对应的值是 b11。对于中间节点，有两个重要域：名字、属性集。如示例中<a>标签，标签名是 a，有 2 个属性，name 属性对应值 a，prop1 对应属性 a2。
- 若同路径下多个叶节点（或中间节点）标签名相同，则可通过设置 name 属性加以区分。如示例中两个<b>标签，一个名字为 b1，另一个名字为 b2，这样就区分开了。

我们的目的是将 XML 文件转化成内存中的树型结构，相应功能类如下。

1. 定义抽象节点类 Node

```
abstract class Node{
    public static final int START_NODE = 0;   //中间节点起点标识
    public static final int END_NODE = 1;     //中间节点终点标识
    public static final int LEAF_NODE = 2;    //叶节点标识
    protected String name;   //名字
    protected Map<String, String> propMap;    //属性集
    protected String value;                   //值
    public static int getNodeType(String s){ //判断节点类型
        int num = 0;
        for(int i=0; i<s.length(); i++){
            if(s.charAt(i) == '<')
                num ++;
        }
        if(num == 1){
            if(s.charAt(1)=='/')
                return END_NODE;
            else
                return START_NODE;
        }
        return LEAF_NODE;
    }
    public void addNode(Node node)throws Exception{
        throw new Exception("Invalid operation");
    }
    public abstract void display();       //显示
}
```

经前文分析，所有节点都有 3 个基本域成员变量：名字 name，值 value，属性集 map 类型变量 propMap。只不过对于中间节点，由于没有值域，故 value 为 null。

getNodeType()判断节点的类型。由于上文对节点做了约定，因此文件中每行数据只能有三种情况：① 读的内容是“<……>”，表明是中间结点的起点；② 读的内容是“</……>”，表明是中间节点的终点；③ 读的内容是“<……>……</……>”，表明是叶节点。getNodetType()方法根据“<”，“>”，“/”这三个字符，可判定读的字符串内容 s 究竟是哪一种节点。

2. 叶子节点类 LeafNode

```
class LeafNode extends Node{
    public LeafNode(String s){    //s 形如<b name=b2>bvalue2</b2>
        int start = 1;
        int end = s.indexOf('>');
```

```
String mid = s.substring(start,end); //拆分结果为[b name=b2]
String mid2[] = mid.split("");
name = mid2[0];                    //姓名, mid2[0]="b" mid2[1]="name=b2"
if(mid2.length>=2)
    propMap = new HashMap();
for(int i=1; i<mid2.length; i++){     //属性
    String mid3[] = mid2[i].split("=");
    propMap.put(mid3[0], mid3[1]);   //mid3[0]=name, mid3[1]=b2
}
start = end+1;
        end = s.indexOf('<',start);
        value = s.substring(start, end);      //值, bvalue2
    }
    public void display(){
        System.out.println(name);
    }
}
```

由于是叶子节点，在构造方法 LeafNode()中，其形参 s 一定是形如“ bvalue2”形式。因此思考好如何拆分该字符串，就能方便得出叶子节点的名字、值、属性值 map 映射。

3. 中间节点类 MidNode

```
class MidNode extends Node{
    ArrayList<Node> nodeList = new ArrayList();
    public MidNode(String s){                    //s形如<a name=a prop1=a2>
        String mid = s.substring(1,s.length()-1);
        String mid2[] = mid.split("");
        name = mid2[0];                          //姓名
        if(mid2.length>=2)
            propMap = new HashMap();
        for(int i=1; i<mid2.length; i++){     //属性
            String mid3[] = mid2[i].split("=");
            propMap.put(mid3[0], mid3[1]);
        }
    }
    public void addNode(Node node){
        nodeList.add(node);
    }
    public void display(){
        System.out.println(name);
        for(int i=0; i<nodeList.size(); i++){
            nodeList.get(i).display();
        }
    }
}
```

由于是中间节点，在构造方法 LeafNode()中，其形参 s 一定是形如“<a name=a prop1=a2>”形式。因此思考好如何拆分该字符串，就能方便得出中间节点的名字、属性值 map 映射。

4. XML 文件解析类 XMLManage

```
class XMLManage{
    Node root;
```

```
    public boolean create(String strFile)throws Exception{/*根据 XML 文件创建树*/}
    Node getLayer(String s[]){/**/}
    String getValue(String strPath,String strPropValue) {/*确定该路径节点对应的值*/}
    String getValue(String strPath, String strProperty) {/*根据路径及属性值确定对应值*/}
    String getProperty(String strPath, String strProp) {/*根据路径及属性名确定属性值*/}

}
```

该类主要包含一个创建树方法 create(),4 个解析树的 getXXX()方法。下面先讲解 create()。

```
    public boolean create(String strFile)throws Exception{
        FileReader in = new FileReader(strFile);
        BufferedReader in2 = new BufferedReader(in);
        String s = in2.readLine();
        s = s.trim();
        root = new MidNode(s);
        Stack<Node> st = new Stack();
        st.push(root);
        while((s=in2.readLine())!=null){
            s = s.trim();
            if(s.equals("")) continue;
            int mark = Node.getNodeType(s);
            if(mark == Node.START_NODE){          //起始节点
                Node node = new MidNode(s);
                Node top = st.peek();
                top.addNode(node);
                st.push(node);
            }
            else if(mark == Node.END_NODE){       //终止节点
                st.pop();
            }
            else{                                 //叶子节点
                Node node = new LeafNode(s);
                Node top = st.peek();
                top.addNode(node);
            }
        }
        in2.close();
        in.close();
        return true;
    }
```

本示例主要采用堆栈对“起始结点”、“叶子节点”、“终止节点”进行处理。总体原则是：当遇到起始节点时进行入栈处理，由于它一定是当前栈顶元素的子节点，因此入栈前把它添加到栈顶元素的子集向量列表中；当遇到终止节点时进行出栈处理；当遇到叶子节点时既不入栈也不出栈，只是把它添加到当前栈顶元素的子节点中。

创建树结束后，主要就是解析树，根据路径获得所需的属性值或值了。如何定义路径呢？例如，示例中的<b>标签路径可以定义为“root/a/b”，<a2>标签可以定义为“root/a2”。也就是说，利用形如“.../.../.../...”定义某标签路径。类 XMLManage 中两个 getValue()方法是用来获取标签值的，getProperty()方法是用来获取属性值的。在某路径中若标签唯一，则用 getValue(String strPath)获得值。例如，若获得示例中<c>标签的值，则用 getValue(“root/a/c”)表示。在某路径中若标签不唯一，则用 getValue(String strPath,String strPropValue)获得值，这

时多个标签中应通过设置 name 属性值加以区分。例如，若获得示例中两个<b>标签对应的值，可分别用 getValue(“root/a/b”, “b1”)、getValue(“root/a/b”, b2) 表示。

有两个 getValue()方法，一个是根据路径确定值，另一个是根据路径+属性值确定值。不同一定是发生在路径倒数第 2 层节点的遍历上，伪码比较如表 12-3 所示。

表 12-3　　后两层遍历情况伪码比较

根据路径确定值	根据路径+属性值确定值
遍历倒数第 2 层节点 　　若子节点路径等于最终路径，则输出值。	遍历倒数第 2 层节点 　　若子节点路径等于最终路径，且 name 属性值等于所给值，则输出值。

对于倒数第 2 层之上的路径遍历都是相同的，因此，定义了 getLayer 方法，用以获得倒数第 2 层的节点对象，如下所示。

```
Node getLayer(String s[]){
    Node result = root;
    try{
        if(!root.name.equals(s[0]))
            return null;
        ArrayList<Node> mid = ((MidNode)root).nodeList;
        int i, j;
        for(i=1; i<s.length-1; i++){
            boolean bmark=false;
            for(j=0; j<mid.size(); j++){
                String name = mid.get(j).name;
                if(name.equals(s[i])){
                    bmark = true; break;
                }
            }
            if(!bmark)
                return null;
            result = mid.get(j);
            mid = ((MidNode)result).nodeList;
        }
    }
    catch(Exception e){
        return null;
    }
    return result;
}
```

方法 getLayer，依次与 s[0]~s[n-2]匹配，即获得所需节点对象。

XMLManage 中，getValue()、getProperty()方法均调用了 getLayer()方法，具体代码如下。

```
String getValue(String strPath){          //根据路径获得值
    String s[] = strPath.split("/");      //拆分获得路径数组
    Node result = getResult(s);           //获得倒数第 2 层节点对象
    if(result==null)
        return null;
    boolean bmark = false;
    ArrayList<Node> mid = ((MidNode)result).nodeList;
    for(int i=0; i<mid.size(); i++){      //遍历第 2 层节点数组
```

```
            result = mid.get(i);
            if(result.name.equals(s[s.length-1])){//若路径匹配
                bmark = true;break;
            }
        }
        if(bmark)     //若有结果
            return result.value;//则返回
        return null;
    }
    String getValue(String strPath, String strPropValue){ //根据路径+属性获得值
        String s[] = strPath.split("/");                    //拆分获得路径数组
        Node result = getResult(s);                         //获得倒数第2层节点对象
        if(result==null)
            return null;
        boolean bmark = false;
        ArrayList<Node> mid = ((MidNode)result).nodeList;
        for(int i=0; i<mid.size(); i++){          //遍历第2层节点数组
            result = mid.get(i);
            Map<String, String> map = result.propMap;//若路径+属性值匹配
            if(result.name.equals(s[s.length-1]) && strPropValue.equals(map.get
                    ("name"))){
                bmark = true;break;
            }
        }
        if(bmark)     //若有结果
            return result.value;//则返回
        return null;
    }
    String getProperty(String strPath, String strProp){ //根据路径,获得其strProp对应
                                                         //的属性值
        String s[] = strPath.split("/");
        Node result = getResult(s);
        if(result==null)
            return null;
        boolean bmark = false;
        ArrayList<Node> mid = ((MidNode)result).nodeList;
        for(int i=0; i<mid.size(); i++){
            result = mid.get(i);
            if(result.name.equals(s[s.length-1])){
                bmark = true;break;
            }
        }
        if(bmark){
            Map<String, String> map = result.propMap;
            return map.get(strProp);
        }
        return null;
    }
```

本示例中的解析功能稍显简单，每次只能返回一个值。其实只要稍加思考，就能容易进行功能拓展。这里只列出功能描述，代码就不列举了，读者可自行完成，如表12-4所示。

表 12-4　　解析功能拓展列表

方法名	说明
Node getNode(String path)	返回 path 路径对应的节点对象
String[] getValue(String prePath, String path[])	一次返回多个标签的值，标签路径分别由 prePath+path[0],…,prePath+path[n]组成
Map<String,String>getAllProperty(String path)	返回标签的所有属性值，返回 map 类型
Map<String,String> getAllProperty(String path, String propValue)	在同名标签中返回 name 属性值等于变量 propValue 值对应标签的所有属性

一个简单的测试类如下所示。

```
public class Test {
    public static void main(String[] args)throws Exception {
        XMLManage obj = new XMLManage();
        obj.create("C:/config.xml");
        System.out.println(obj.getValue("root/a2"));
        System.out.println(obj.getValue("root/a/b","b1"));
        System.out.println(obj.getValue("root/a/b","b2"));
        System.out.println(obj.getProperty("root/a","prop1"));
    }
}
```

总之，编制自定义解析 XML 类的过程并不是太复杂。笔者认为，目前众多的 XML 解析器最大的特点是通用性，但一般来说是以牺牲时间为代价的。而我们的应用程序不论大小，或多或少都有它的特殊性，对配置文件来说也一样。因此编制适合我们应用的自定义 XML 文件解析器也是非常有价值的，更主要的是在编程过程中思路会不断涌现，错误、正确的解析结果交替出现，因此非常锻炼思维能力，增加编程的乐趣。

【例 12-3】 HTML 的框架集标签<frameset>用于把 Web 页面划分成不同的部分，每一个部分可以显示一个独立的 Web 页面。使用框架标签<frame>可以让一个实际的 Web 页面在网页的一个部分中显示出来。而且，<frameset>标签可以是嵌套的，一个具体的框架示例如下。

```
<frameset rows="20%,80%">
    <frameset rows="100,200">
        <frame src="frame1.html">
        <frame src="frame2.html">
    </frameset>
    <frame src="frame3.html">
</frameset>
```

可以看出，<frameset>、<frame>可以看作树中的节点。前者是中间节点，后者是叶子节点。

要求：设计两个类 FrameSet 和 Frame 来分别表示<frameset>和<frame>标签；在类中定义一个方法，用于返回 FrameSet 或者 Frame 对象特定的 HTML 文件；通过组合模式设计并且实现该操作，使客户程序可以没有区别地使用 FrameSet 类和 Frame 类。

1. 定义界面抽象节点类 UINode

```
abstract class UINode{
    String s;
    public UINode(String s){
```

```
            this.s = s;
        }
        public void addNode(UINode node)throws Exception{
            throw new Exception("Invalid exception");
        }
        abstract void display();
}
```

成员变量 s 对 Frame 叶子子对象而言，代表 HTML 文件；对 FrameSet 中间节点子对象而言，代表普通的名字。

2. 定义叶子类 Frame

```
class Frame extends UINode{
    public Frame(String strPath){ //strPath 表示 html 文件路径
        super(strPath);
    }
    public void display(){
        System.out.println(s);
    }
}
```

3. 定义中间节点类 FrameSet

```
class FrameSet extends UINode{
    private ArrayList<UINode> nodeList = new ArrayList();
    public FrameSet(String name){
        super(name);
    }
    public void addNode(UINode node)throws Exception{
        nodeList.add(node);
    }
    public void display(){
        for(int i=0; i<nodeList.size(); i++){
            nodeList.get(i).display();
        }
    }
}
```

在 display()方法中不要在循环前添加“System.out.println(s)”语句，题意要求仅显示叶节点的 HTML 文件名称即可。

一个简单的测试类如下（与题中所给示例一致）。

```
public class Test {
    public static void main(String[] args)throws Exception{
        FrameSet root = new FrameSet("1");    //中间节点名称
        FrameSet child = new FrameSet("2");
        Frame frm1 = new Frame("frame1.html");    //必须是 html 文件
        Frame frm2 = new Frame("frame2.html");
        Frame frm3 = new Frame("frame3.html");
        child.addNode(frm1); child.addNode(frm2);
        root.addNode(child); root.addNode(frm3);
        root.display();  //中间节点调用 display()显示所需 html 文件
        frm1.display();  //叶节点也调用 display()显示所需 html 文件，形式一致
    }
}
```

参 考 文 献

[1] 陈臣，王彬. 研磨设计模式[M]. 北京：清华大学出版社，2011.

[2] 张逸. 软件设计精要与模式[M]. 北京：电子工业出版社，2010.

[3] Partha Kuchana. Java 软件体系结构设计模式标准指南[M]. 王卫军，楚宁志，译. 北京：电子工业出版社，2006.

[4] Eric T Freeman. Head First 设计模式（中文版）[M]. Oreily Taiwan 公司，译. 北京：中国电力出版社，2007.

[5] Erich Gamma. 设计模式——可复用面向对象软件的基础[M]. 李爱军，译. 北京：机械工业出版社，2005.

[6] 耿祥义. Java 设计模式[M]. 北京：清华大学出版社，2009.

[7] 刘径舟，张玉华. 设计模式其实很简单[M]. 北京：清华大学出版社，2013.

[8] 刘伟. 设计模式实训教程[M]. 北京：清华大学出版社，2012 年 1 月。

[9] James W. Cooper，C#设计模式[M]. 叶斌，译. 北京：科学出版社，2011.

[10] John Vlissides. 设计模式沉思录[M]. 葛子昂，译. 北京：人民邮电出版社，2010.

[11] Alan Shalloway, James R. Trott. 徐言声，译. 北京：人民邮电出版社，2013.

[12] 秦小波. 设计模式之禅[M]. 北京：机械工业出版社，2010.

[13] Kirk Knoernschild. Java 应用架构设计[M]. 张卫滨，译. 北京：机械工业出版社，2013.

[14] Y.Daniel Liang. Java 语言程序设计（基础篇）[M]. 李娜，译. 北京：机械工业出版社，2011.

[15] 柳永坡，刘雪梅，赵长海. JSP 应用开发技术[M]. 北京：人民邮电出版社，2005.

[16] Bruce Eckel,ChuckAllison. C++编程思想第 2 卷-实用编程技术[M]. 刁成嘉，等，译. 北京：机械工业出版社，2006.